Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Aplicación Android de grupos de entrenamiento P2P

Autor: Ricardo Martín Muñoz
Tutora: Dra. María Teresa Ariza Gómez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Departamento de Ingeniería Telemática

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Aplicación Android de grupos de entrenamiento P2P

Autor:
Ricardo Martín Muñoz

Tutora:
Dra. Mª Teresa Ariza Gómez
Profesor titular

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Autor: Ricardo Martín Muñoz

Tutora: Dra. María Teresa Ariza Gómez

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

El planteamiento, desarrollo y elaboración de este Trabajo Fin de Grado no habría sido posible sin la contribución de varias personas que han pavimentado el largo pero satisfactorio camino que me ha llevado hasta el final de este Grado en Ingeniería de las Tecnologías de Telecomunicación. Es por ello que quiero aprovechar este espacio para plasmar mi gratitud a dichas personas.

En primer lugar, y como ha sido desde el primer momento de mi vida, mis padres. Gracias por creer en mí incluso cuando he tropezado tantas, tantas veces. Gracias por vuestro apoyo y gracias también por vuestra severidad que me ha enseñado que para obtener resultados hay que realizar un esfuerzo. Gracias por instruirme y educarme convirtiéndome en la persona que soy hoy. Aunque a veces no haya dado la talla sé que lo que está por venir lo podré afrontar gracias a todo lo que me habéis dado. Gracias.

También gracias a mi hermano y a mi hermana. Todo este tiempo me han hecho ver la otra cara de la educación al ser yo el que por una vez debía enseñar en lugar de ser enseñado. Sé que ambos serán brillantes profesionales y su apoyo, cariño y cercanía es una de las razones por las que trato de avanzar siempre. Gracias.

Gracias, a continuación, a mis profesores por todo el tiempo que han empleado en clase y fuera de ella en mi formación. En especial, gracias a Juan Manuel Vozmediano, quien me impartió la asignatura que más quebraderos de cabeza y trabajo me ha costado de toda la carrera sin por ello dejar de parecerme un profesor remarcable y ganarse mi simpatía; a Francisco Javier Muñoz Calle, el cual ha sido, además de un gran profesor, una figura muy cercana por todo su apoyo y comprensión; y a María Teresa Ariza Gómez, mi tutora de este Trabajo Fin de Grado y profesora de varias de las asignaturas que más he disfrutado de la carrera y me hicieron decidirme por la Telemática como especialidad. Gracias.

No quiero olvidar tampoco a todos los compañeros de piso, de clase y amigos en general que han hecho más llevadera esa etapa de mi vida. Bien del instituto o bien ya en la universidad, las personas que he conocido me han hecho disfrutar plenamente de este periodo. Gracias.

Finalmente, gracias a mi pareja por todo el apoyo que me ofrece ante cualquier dificultad que encuentre. Significa mucho para mi saber que estás ahí en todo momento y que puedo contar contigo para lo que sea. Pese a haber afrontado muchas dificultades juntos siempre salimos adelante, y este trabajo no es si no otra prueba más de ello. Gracias.

Ricardo Martín Muñoz

Sevilla, 2016

Resumen

La interacción es uno de los objetivos principales de las aplicaciones móviles en la actualidad. Se conectan con juegos, con otras aplicaciones y con sistemas de almacenamiento remotos, ya que los dispositivos móviles se han convertido, en última instancia, en una herramienta no solo de comunicación si no de interacción. Ya no solo hablamos por teléfono, ahora mandamos imágenes, vídeo y archivos, indicamos dónde estamos o cuáles son nuestros planes, dónde vamos a comer o qué película acabamos de ver. Y, por qué no, qué ejercicio vamos a hacer,

En este trabajo se ha desarrollado una aplicación para que los usuarios puedan crear grupos de entrenamientos y anunciarlos a otros usuarios para hacer ejercicios juntos. Varias personas pueden unirse y chatear mientras controlan la reproducción de los ejercicios que deben ir realizando, intercambiando dudas o sugerencias en el proceso si lo deseasen. Además, estos grupos se enlazan entre sí directamente en una red P2P, por lo que no es necesaria la presencia de un servidor y los usuarios pueden confiar en que la aplicación estará siempre disponible sin caídas del servicio.

El objetivo principal de este trabajo ha sido, por tanto, la implementación de la tecnología P2P para crear un entorno de entrenamiento grupal con reproducción de vídeo.

Abstract

Interaction is one of the main objectives of mobile apps nowadays. They stablish connections with games, people and remote storage services because mobile devices have become ultimately, in a tool not just for communication but also for interaction. We don't only speak anymore, now we send images, videos and files, we show where we are or which are our plans, where are we going to eat or what movie we just watched. And thus, why not, the training we are going to do.

This project develops an application for users to create and announce training groups to other users so they can exercise together. Multiple people can join and chat while they control the video of the exercises they perform, sharing thoughts and doubts if they will. Also these groups are shared directly in a P2P network, which makes unnecessary the presence of a server, allowing the users to access the app anytime without downtimes.

Therefore the main objective of this project has been the implementation of P2P technology to create a training environment with video reproduction.

Índice

Índice de Figuras

1 INTRODUCCIÓN

Este primer capítulo ilustra los motivos que han llevado a la elaboración de este trabajo y el alcance de la solución ofrecida, así como un esbozo de los elementos que toman parte en ella y su arquitectura. Se presentan también los objetivos tenidos en cuenta a la hora de realizar el trabajo y los antecedentes similares a la aplicación desarrollada.

1.1 Motivación

Llevar un estilo de vida sano es una de las preocupaciones recurrentes de la sociedad moderna. Desde la alimentación a la salubridad del aire, todos los factores que intervienen en la salud son estudiados en mayor o menor medida para tratar de mantener el cuerpo en las mejores condiciones posibles.

El ejercicio es uno de los elementos relevantes de estos factores, influyendo de manera directa y evidente en la forma física de las personas, y como tal ha sido objeto de promoción, marketing y venta de muchas maneras distintas a lo largo del tiempo. Tradicionalmente el formato popular de la comercialización de ejercicio han sido los gimnasios y los productos deportivos (bicicletas, pesas, ropa deportiva…), y todo ello acompañado de la guía y tutelaje de un profesional en la materia.

Hoy en día sin embargo tenemos más recursos a nuestra disposición, entre ellos los dispositivos móviles. Estos nos permiten conectar con otras personas y acceder a enormes cantidades de información que pueden suplir a los expertos en la materia si son usados de manera adecuada.

La aplicación desarrollada en este trabajo pretende, precisamente, llenar ese hueco. Los usuarios podrán llevar a cabo ejercicios complejos de manera guiada, gracias a los vídeos explicativos, sin la necesidad de un monitor que les guíe. Varios amigos podrían quedar juntos para llevar a cabo rutinas de enteramiento, de manera sencilla y cómoda, sin necesitar nada más que sus dispositivos móviles. O por el contrario, un usuario podría usar la aplicación en solitario para ntrenar por sí mismo si lo desease.

Se busca, por tanto, el acercamiento de los usuarios de a pie a un estilo de vida sano, facilitándoles una aplicación de entrenamiento sencilla, intuitiva y de carácter social por su enfoque a grupos.

1.2 Presentación del problema

Las aplicaciones de entrenamiento no son una novedad en Google Play Store. Muchas de ellas gozan de mucha popularidad y un elevado número de descargas. Esto se debe al creciente interés del ejercicio como fuente de una vida saludable y un aumento del atractivo personal, sumado a lo sencillo y libre que es entrenar mediante una aplicación en lugar de tener que desplazarnos hasta un gimnasio.

Sin embargo, la gran mayoría de estas aplicaciones comparten una característica común: están orientadas a la práctica del ejercicio de manera individual. Si bien es cierto que muchas dan la opción de compartir nuestros resultados a través de redes sociales y compararlos con los de nuestros conocidos, en general, ninguna ofrece la opción de hacer ejercicio en grupo. Son de uso individual y conectan al usuario con un servidor central que, a su vez, se comunica con una base de datos como se representa en la figura (1.1).

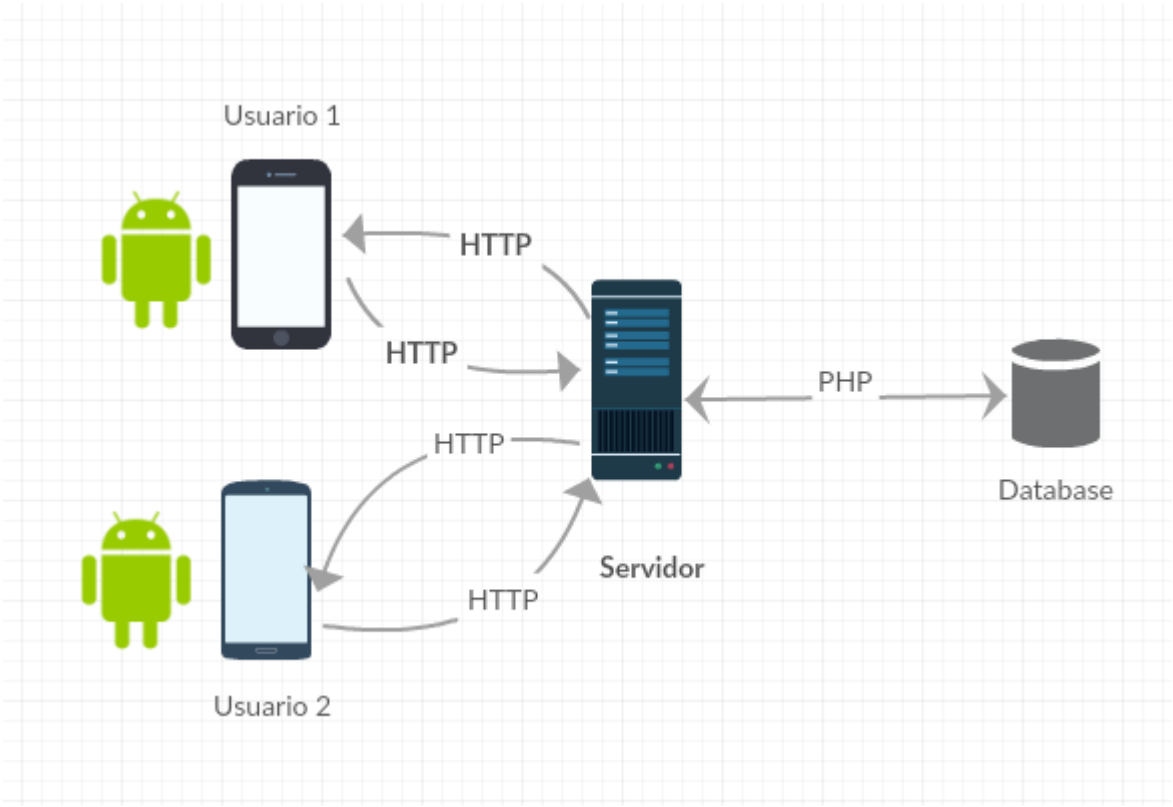

Figura 1.1 Diagrama de comunicación de aplicación con servidor

Con la opción de trabajar en grupo con otros usuarios se abre la posibilidad de crear un ambiente de entrenamiento similar al del gimnasio: varias personas realizando simultáneamente el mismo ejercicio siguiendo un ejemplo de guía. Además, la componente social del ejercicio en grupo diferencia esta aplicación de otras muchas expuestas en el siguiente apartado. Es por eso que el principal objetivo a afrontar durante el desarrollo de este trabajo ha sido realizar una aplicación grupal, en la que varios usuarios puedan interactuar entre sí.

Además, con la idea de eliminar la presencia de un servidor, el desarrollo de esta aplicación se ha llevado a cabo siguiendo la arquitectura peer-to-peer (P2P), que permite a los dispositivos conectar directamente entre ellos sin pasar por un servidor intermediario, como se representa en la figura (1.2). Esto garantiza que la aplicación no sufrirá de tiempos de suspensión por la baja de un servidor y permite, además, a los dispositivos conectar entre ellos aunque no dispongan de un plan de datos móviles o una conexión a internet. Estos aspectos serán explicados y desarrollados más adelante.

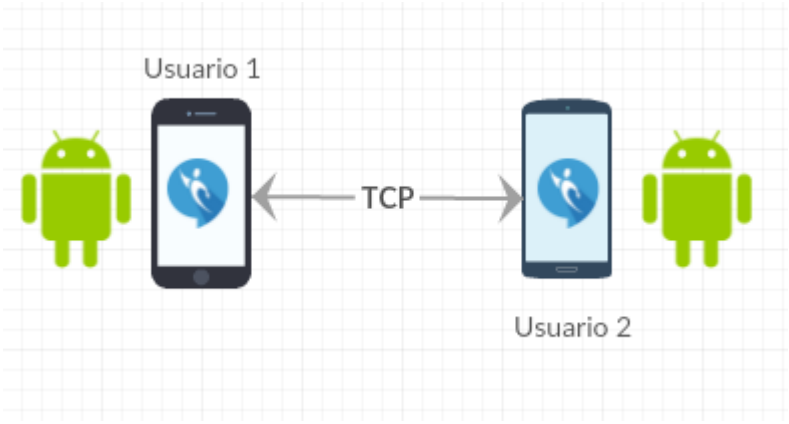

Figura 1.2 Diagrama de comunicación de aplicación peer-to-peer

Como último objetivo a resolver, se decidió que la aplicación debía contar con un historial de ejercicios que permita al usuario mantener un control del ejercicio que va realizando para poder realizar un seguimiento de su progreso y sus actividades favoritas. Para ello, se dispondrá de una base de datos local en la que se irá almacenando la actividad del usuario.

1.3 Soluciones similares

Siendo un producto que vende, el ejercicio autónomo ha sido llevado al ámbito comercial informático en forma de gran cantidad de aplicaciones diversas que se pueden encontrar en Google Play Store y otros servicios de descarga oficial de aplicaciones para diversos sistemas operativos. Sin embargo, como antes se ha mencionado la enorme mayoría de estas aplicaciones están pensadas para su uso individual.

A continuación, se exponen brevemente algunas de las aplicaciones de entrenamiento más populares de Google Play.

1.3.1 Runtastic GPS Running, Fitness

Figura 1.3 Icono de Runtastic GPS Running, Fitness

Sin ofrecer una rutina de ejercicios completa, esta aplicación gratuita se centra solo en una de las variantes más populares del ejercicio amateur individual: la conocida como ‘running’, o en español, correr. Cuenta, además, con algunas variaciones similares como ciclismo.

Esta aplicación hace uso del GPS del móvil y el servicio de mapas de Google Maps para realizar un control de las rutinas de carrera del usuario, midiendo la distancia recorrida, el tiempo empleado en ello y varios factores adicionales como la velocidad media, máxima o una estimación de las calorías consumidas.

El modelo de uso consiste en activar la aplicación al salir a correr y llevar el móvil o algún ‘wearable’ como un ‘smartwatch’ para hacer el seguimiento de la carrera. Además, la aplicación propone diversos retos como completar ciertas distancias en cierto tiempo o batir nuestros resultados anteriores.

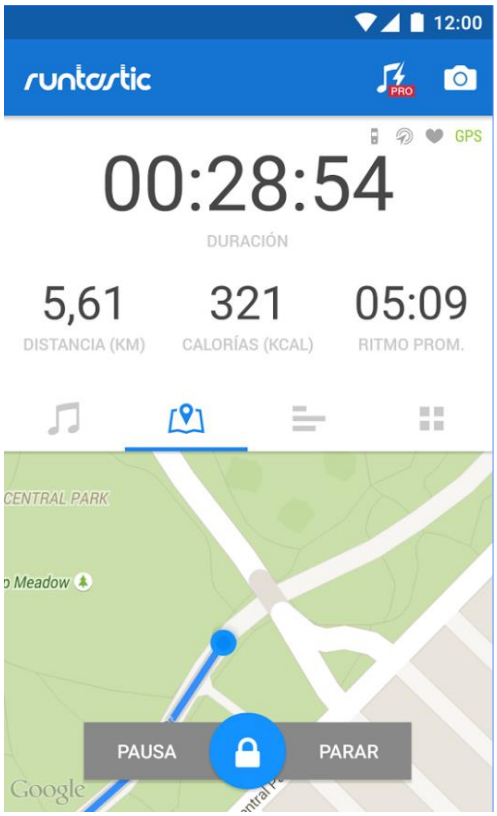

Figura 1.4 Interfaz de usuario en uso de Runtastic GPS Running, Fitness

Figura 1.5 Interfaz de historial de Runtastic GPS Running, Fitness

Además, la aplicación cuenta con la componente social habitual consistente en la posibilidad de compartir los resultados del usuario a través de redes sociales y a través de la propia aplicación, accediendo a un ranking en el que se compite con el resto de usuarios.

Runtastic cuenta con una versión de pago con un precio de 4,99 euros llamada Runtastic PRO Running, Fitness que añade diversas funcionalidades como feedback de audio personalizado para los ejercicios, la posibilidad de ver nuestras rutas en un mapa 3D, gráficas detalladas de elevación, ritmo, frecuencia cardíaca, etc.

1.3.2 Fitness & Bodybuilding

Figura 1.6 Icono de Fitness & Bodybuilding

Alejándose del concepto de correr exclusivamente, esta aplicación, también gratuita, ofrece un amplio conjunto de ejercicios de musculación y fitness en un formato atractivo y ordenado por conjuntos musculares. Las figuras (1.7), (1.8) y (1.9) recogen diversos apartados de la interfaz de usuario.

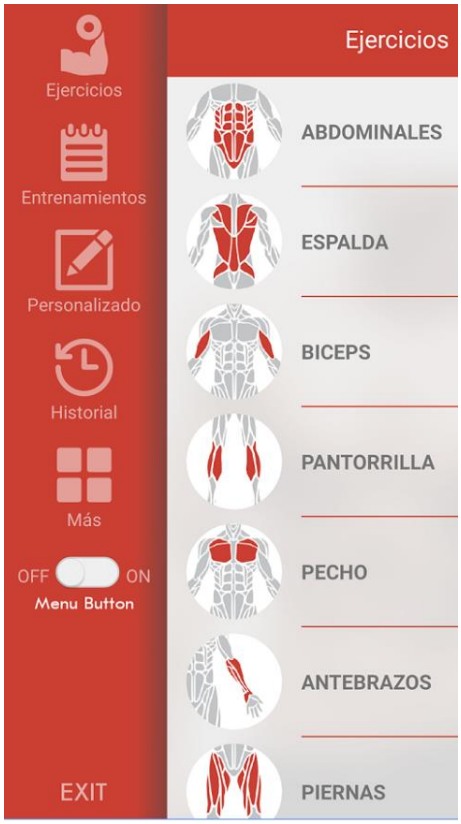

Figura 1.7 Pestaña de selección de grupo muscular

Figura 1.8 Pantalla de descripción de ejercicio

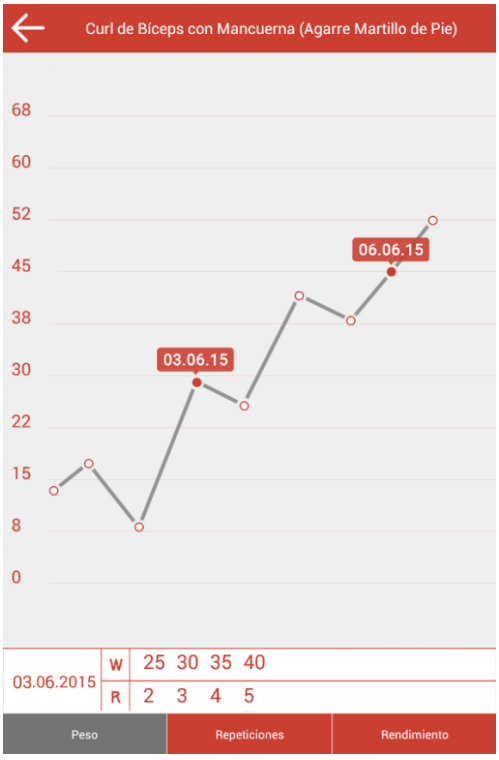

Figura 1.9 Pantalla de historial de ejercicio

Es fácil apreciar que Fitness & Bodybuilding cuenta con un abanico de ejercicios mucho mayor que Runtastic, ya que, al contrario que la anterior, esta sí es una aplicación pensada para el entrenamiento completo del usuario, no solo para el aspecto de carrera.

La descripción de los ejercicios se lleva a cabo mediante imágenes como se observa en la figura (1.8), por lo que el peso de la aplicación es bastante ligero.

1.3.3 Reto Deportivo de 30 Días

Figura 1.10 Icono de Reto Deportivo de 30 Días

Esta aplicación lleva un paso más lejos la realización de ejercicio, permitiendo al usuario establecer un plan de entrenamiento de 30 días de duración, aumentando progresivamente el nivel con el objetivo de obtener resultados en un tiempo relativamente reducido.

La aplicación permite tanto crear nuestro propio plan de entrenamiento como seguir uno ya creado, tal como se refleja en la figura (1.11).

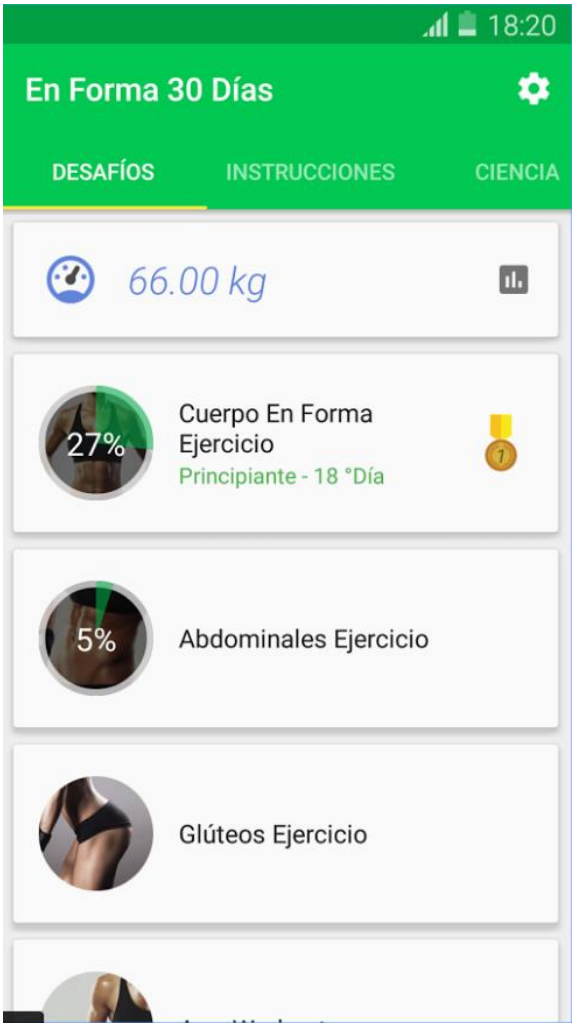

Figura 1.11 Pantalla de selección de planes de entrenamiento prefabricados

La descripción de los ejercicios viene expuesta en forma de vídeos cortos con una breve descripción en inglés.

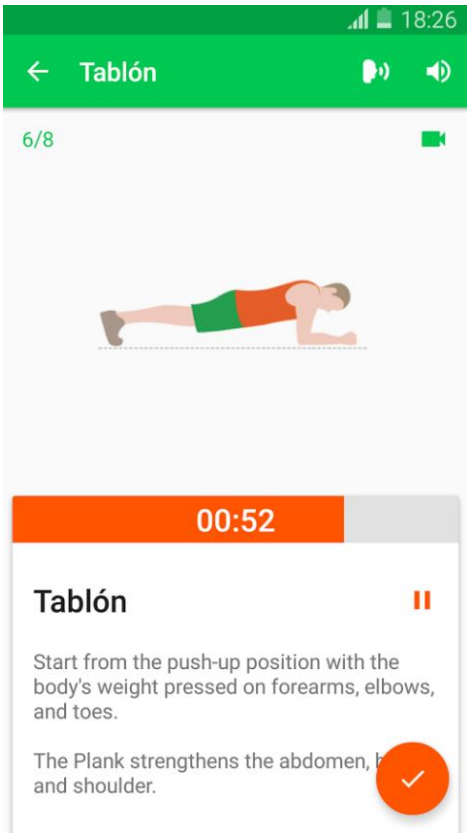

Figura 1.12 Descripción del ejercicio

Además, del mismo modo que la mayoría de aplicaciones, cuenta con un histórico que mantiene un control de los ejercicios realizados y los objetivos que se van cumpliendo (número de repeticiones, peso perdido, volumen de ejercicio realizado…). Y, como es común, permite compartir dichos resultados a través de las redes sociales.

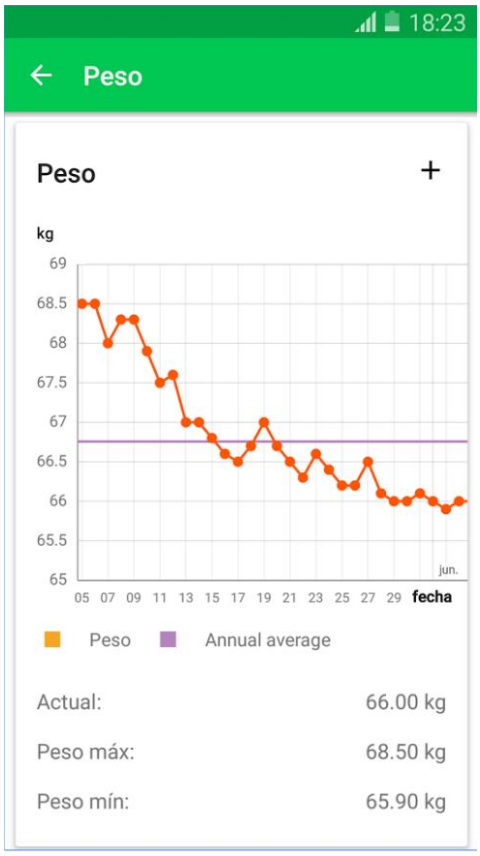

Figura 1.13 Gráfico de progreso

1.4 Descripción de la solución propuesta

La solución propuesta en este trabajo para la aplicación de entrenamiento ha recibido el nombre de P2PTrain. Dicho nombre ha sido escogido con base en el principal punto diferenciador de la solución respecto a otras aplicaciones de ejercitación personal: el formato grupal de la misma obtenido a partir de la creación de conexiones de tecnología peer-to-peer (P2P) con otros usuarios.

P2PTrain es una aplicación móvil para Android que permite, de manera sencilla para los usuarios, crear y gestionar grupos de entrenamiento físico en los que conversar con otros usuarios mientras controlan la reproducción de vídeos explicativos acerca de la realización de distintos ejercicios. Para implementar este entorno P2P se han utilizado las librerías de AllJoyn, tecnología de AllSeen Alliance[1]. Haciendo uso de ellas de manera transparente para el usuario este puede conectar con otros dispositivos y compartir rutinas de entrenamiento, aunque si lo prefiere también es libre de entrenar individualmente.

El usuario puede, asimismo, unirse a grupos creados por otros usuarios con la misma finalidad, los cuales son anunciados mediante difusión multicast. El proceso detallado está explicado en apartados posteriores de este documento.

Además, los ejercicios que realice el usuario quedan registrados en una base de datos local de la aplicación implementada mediante SQLite[2]. Mediante consultas a esta base de datos se muestran por pantalla las rutinas llevadas a cabo junto con la fecha y el grupo en el que se realizaron.

En conjunto P2PTrain conforma una aplicación de usuario sencilla e intuitiva mediante la que realizar ejercicio en grupo, el cual es el objetivo principal de este trabajo.

[1] https://allseenalliance.org/
[2] https://sqlite.org/

1.5 Agentes del sistema

El único agente que compone el sistema de este trabajo es la **Aplicación Android** (P2PTrain) que utilizarán los usuarios en sus dispositivos móviles. Esta es compatible con dispositivos móviles que utilicen una versión de Android igual o superior a la 4.1, la llamada Jelly Bean, debido a las necesidades de las librerías de AllJoyn implementadas.

Como subagentes de la Aplicación Android se puedn identificar la **Base de datos** de la aplicación, encargada de contener el historial de ejercicios realizados, y el **Módulo de AllJoyn** encargado de realizar las comunicaciones de la aplicación.

En un futuro podría incluirse un servidor para gestionar el acceso de los usuarios a la aplicación. Esta opción se discute en el punto 6.1 de este documento.

2 TECNOLOGÍAS EMPLEADAS

Este segundo capítulo describe de manera detallada el entorno de desarrollo empleado para la realización de este trabajo, explicando pormenorizadamente cada uno de los elementos y herramientas implicadas en el proceso.

2.1 Android

Android es un sistema operativo ideado en su origen para teléfonos móviles del mismo modo que iOS, Windows Phone o Blackberry. A diferencia de estos, Android está basado en el núcleo del sistema operativo libre Linux, el cual es gratuito y multiplataforma.

Figura 2.1 Logotipo de Android

El S.O. Android proporciona las interfaces necesarias para desarrollar aplicaciones que pueden acceder a funciones del teléfono (como el GPS, la agenda de contactos…), de manera sencilla, a través del lenguaje de programación Java.

Android fue desarrollado por Android Inc, estudio prácticamente desconocido hasta que Google lo adquirió en 2005. Ese mismo año comienzan a trabajar en la creación de una máquina virtual Java optimizada para dispositivos móviles. En el año 2007 se crea el consorcio Handset Alliance, grupo formado por varias empresas y compañías, entre ellas Google, con el fin de desarrollar estándares de telefonía móvil. Uno de sus objetivos era promover el diseño y la difusión de la incipiente plataforma Android.

En noviembre del 2007 se lanza una primera versión del Android SDK. Al año siguiente apareció el primer teléfono con Android (T-Mobile G1). En octubre Google libera el código fuente de Android bajo licencia de código abierto Apache. Ese mismo mes se abre Android Market para la descarga de aplicaciones. En 2009 se lanza la versión 1.5 del SDK que incorpora el teclado en pantalla. A finales del 2009 se lanza la versión 2.0 y durante el 2010 las versiones 2.1, 2.2 y 2.3.

En el año 2010 llegaría la consolidación de Android, como uno de los sistemas operativos móviles más usados.

En 2011 se lanzan las versiones 3.0, 3.1 y 3.2 específica para tablets y la 4.0, tanto para móviles como para tablets. Durante este año Android se consolida como la plataforma para móviles más importante, alcanzando cuotas de mercado del 50%.

En 2012 Google reemplaza su Android Market por Google Play Store y se lanzan las versiones 4.1 y 4.2 del SDK. Android mantiene su espectacular crecimiento alcanzando cuotas de mercado del 70%.

En 2013 se lanzan las versiones 4.3 y 4.4 y en 2014 se lanza la versión 5.0 (Lollipop). A finales de este año la cuota de mercado alcanza el 85%.

La versión más reciente de Android, la 6.0, fue lanzada en marzo del 2015 y recibió el nombre de Marshmallow.

Android presenta una serie de características que lo hacen diferente de otros sistemas operativos, combinando en una misma solución las siguientes cualidades:

- Código abierto:

  Android es una plataforma de desarrollo libre basada en Linux y de código abierto, lo que permite usar, copiar e incluso modificar el sistema sin pagar derechos de autor.

- Independencia del hardware:

  Android permite programar sus aplicaciones de manera que pueden ser utilizadas en multitud de dispositivos diferentes: smartphones, tablets, wearables (como relojes inteligentes), televisiones inteligentes… Esta adaptabilidad requiere un mayor esfuerzo a la hora de programar respecto a otros sistemas, pero ofrece una mayor flexibilidad a cambio.

- Portabilidad:

  El lenguaje de programación Java es interpretado mediante una máquina virtual, lo que permite que las aplicaciones puedan ejecutarse en cualquier CPU ya que no dependen de una arquitectura física específica.

- Seguridad efectiva:

  Cada aplicación recibe una serie de permisos que limitan el espectro de acciones que puede realizar en el dispositivo. Además, al ejecutarse las aplicaciones siguen un modelo de computación aislada como en Linux, apartándolas del resto del sistema.

- Optimización para escasos recursos:

  Android utiliza la máquina virtual Dalvik. Se trata de una implementación de Google de la máquina virtual de Java optimizada para dispositivos móviles.

- Gráficos y sonido de calidad:

  Haciendo uso de los códecs de audio y vídeo más comunes y empleando gráficos vectoriales suavizados Android, ofrece una experiencia audiovisual de gran calidad al usuario.

- Arquitectura de componentes:

  La interfaz de usuario se realiza en xml, lenguaje ampliamente empleado en el desarrollo de aplicaciones web.

- Gran cantidad de servicios incorporados:

  Las librerías nativas de Andoid permiten hacer uso de manera sencilla de múltiples funcionalidades de los dispositivos móviles como GPS, navegador, reproductor multimedia...

2.2 Android Studio

Figura 2.2 Logo de Android Studio

Android Studio es el IDE (Integrated Development Environement, o Entorno de Desarrollo Integrado en español) desarrollado por Google utilizado en este trabajo para la creación de la aplicación Android. La primera versión estable de Android Studio se lanzó en diciembre de 2014, momento en el que sustituyó oficialmente a Eclipse como IDE de Android.

Publicado gratuitamente a través de la Licencia Apache 2.0[3], Android Studio está disponible para las plataformas Microsoft Windows, Mac OS X y GNU/Linux.

Es un entorno optimizado para el desarrollo de Android y ofrece una gran variedad de herramientas y opciones tales como:

- Renderización en tiempo real de la aplicación, expuesta en la figura (2.3)
- Construcción basada en Gradle[4], herramienta de automatización de construcción de código.
- Arreglos rápidos para errores de código.
- Plantillas para crear diseños comunes de Android.
- Soporte para programar aplicaciones para Android Wear, versión de Android para dispositivos wearables como relojes inteligentes.
- Emulación en dispositivo virtual o físico de las aplicaciones desarrolladas.

[3] https://es.wikipedia.org/wiki/Apache_License
[4] https://gradle.com/

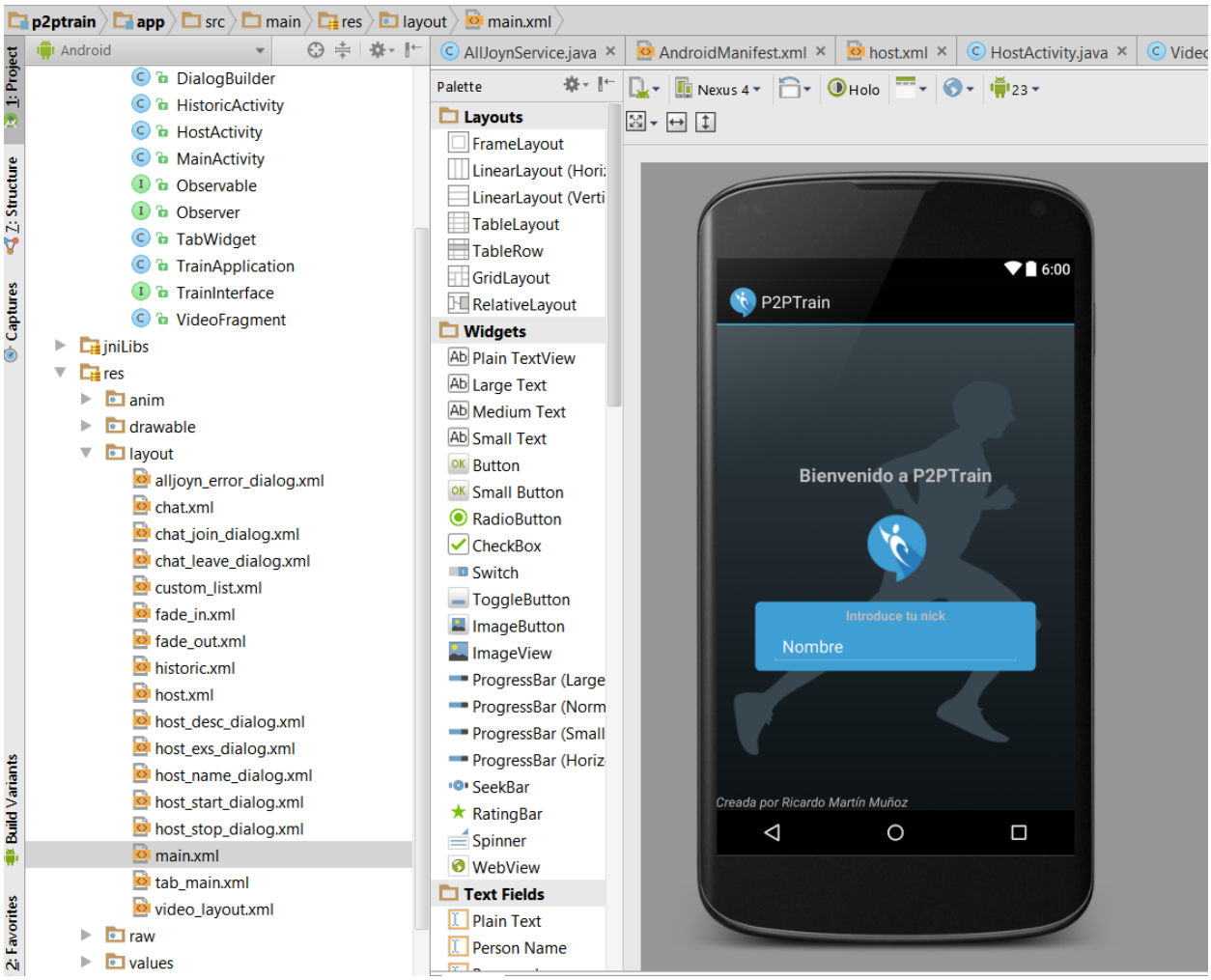

Figura 2.3 Interfaz gráfica de Android Studio

2.3 SQLite

SQLite es una solución embebida y muy ligera (~700kiB[5]) para Android que actúa como gestor de bases de datos relacionales.

Figura 2.4 Logotipo de SQLite

Las características que ofrece SQLite la convierte en una librería muy adecuada para aplicaciones Android. Entre ellas destacan:

- Rápida velocidad de ejecución debido al escaso uso de memoria del dispositivo.
- Formato multiplataforma de base de datos, lo que permite utilizarlas independientemente de si el sistema es de 32 o 64 bits.
- No tiene dependencias externas.

[5] https://sqlite.org/download.html

- Contiene las bases de datos en un único archivo, sin necesidad de un proceso servidor adicional.
- Muy reducido tamaño y uso de memoria en la pila de procesos.
- Código fuente público y extensamente documentado.

En este trabajo SQLite se utiliza a través de la clase DBHandler, detallada en el Apartado 6.1 y el Anexo 3 de este documento.

2.4 AllJoyn

Figura 2.5 Logotipo de AllJoyn

AllJoyn constituye el núcleo de P2PTrain, actuando como punto de comunicaciones de la aplicación local con otros dispositivos remotos. La versión utilizada en este trabajo es la más reciente de AllJoyn: 15.09.

AllJoyn es un conjunto de librerías desarrolladas por AllSeen Alliance, un "consorcio inter-industrial dedicado a permitir la interoperabilidad de billones de dispositivos, servicios y aplicaciones que conforman el Internet de las Cosas", tal y como se definen en la portada de su web[6].

Estas librerías tienen como objetivo permitir la conexión peer-to-peer de distintos dispositivos y aplicaciones de manera uniforme. Por ejemplo, en su web muestran el uso de AllJoyn para conectar un dispositivo móvil con un altavoz inalámbrico y con el sistema de audio de un coche a través de la misma interfaz.

Para lograr esto AllJoyn utiliza una estructura de bus multicast y un sistema de descubrimiento de servicios que puede o bien ser basada en nombres únicos (wellknown-name) o en anuncios. En P2PTrain se implementa el descubrimiento basado en nombres únicos.

El proceso de comunicación consta, pues, de dos fases. En la primera, nuestra aplicación crea una conexión al bus multicast atendiendo en la dirección multicast de difusión de AllJoyn, 224.0.0.251:5353[7]. Una vez está escuchando en esta dirección, comienza el servicio de descubrimiento de nombres, el cual se refleja en la figura (2.6) obtenida directamente de la documentación de AllSeen Alliance[8], así como el resto de figuras empleadas en este apartado.

[6] https://allseenalliance.org/
[7] https://lists.allseenalliance.org/pipermail/allseen-core/2015-September/002479.html
[8] https://allseenalliance.org/framework/documentation/learn/core/system-description

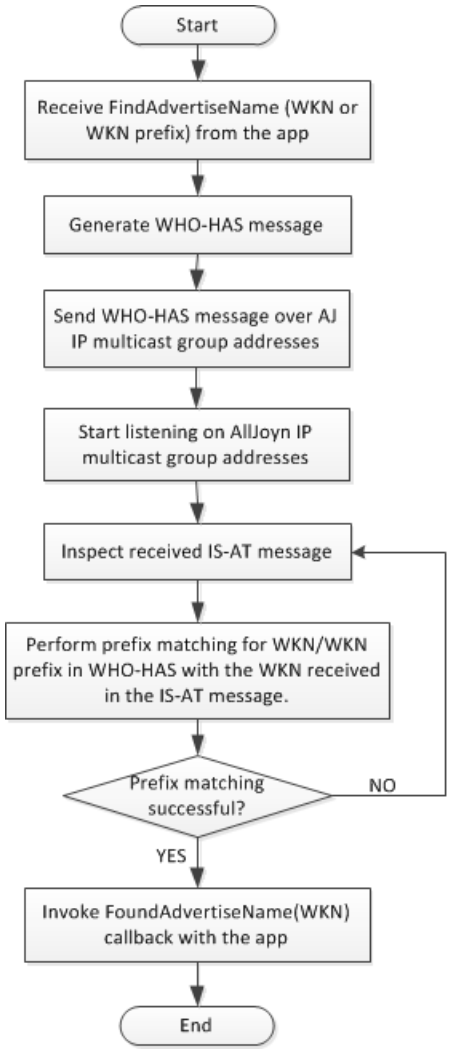

Figura 2.6 Proceso de descubrimiento mediante nombre

Como se ve en este diagrama de flujo, lo primero que la aplicación requiere es un WKN (Well Known Name), un nombre identificativo, que va incluido en el código de la misma. Para P2PTrain este nombre es "org.proyecto.ricardo.p2ptrain", detallado en el Anexo 3 de este documento.

Una vez definido el nombre, envía un mensaje sobre la dirección multicast preguntando quién comparte dicho nombre y queda a la espera de respuesta, y al recibirla comprueba si el nombre recibido coincide con el preguntado. De ser así comunica el resultado exitoso y se procede con el resto de la aplicación, y en caso contrario queda de nuevo a la espera.

Tras la confirmación de descubrimiento del nombre, se establece el bus entre los usuarios que se vayan uniendo. Cada uno tiene una dirección propia en el bus como se muestra en la figura (2.7).

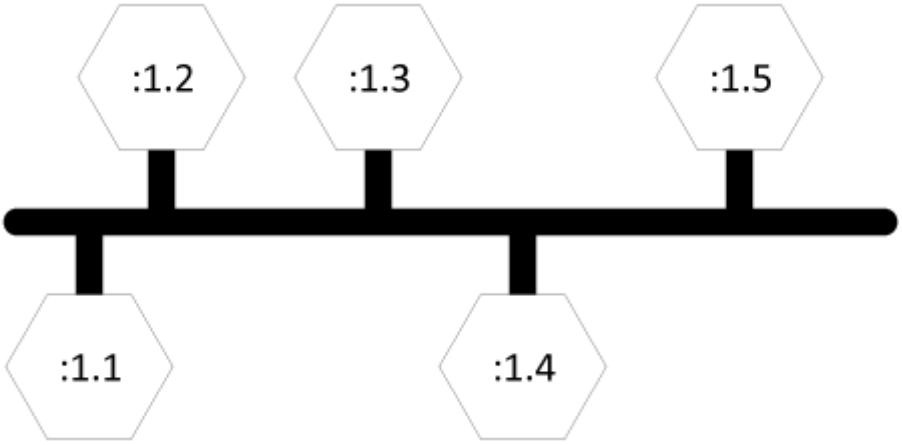

Figura 2.7 Bus AllJoyn con diversos usuarios incorporados

Un mismo bus, además, puede ser compartido por distintas aplicaciones de un mismo dispositivo, permitiéndolas interactuar entre sí y con aplicaciones pares remotas, creando así un bus distribuido. La figura (2.8) refleja este escenario.

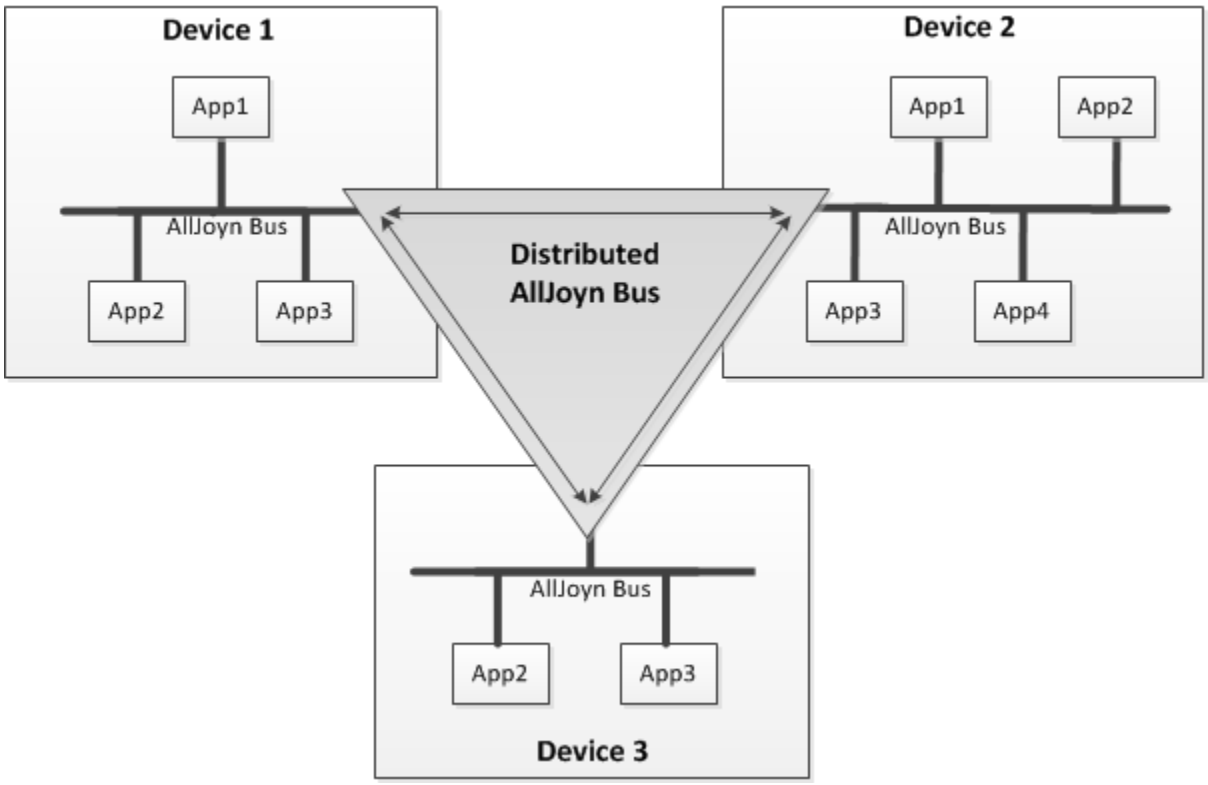

Figura 2.8 Bus de múltiples aplicaciones

Así varias aplicaciones con interfaces o utilidades diferentes podrían compartir información de manera simultánea. Por ejemplo, si se dispusiese de una aplicación que controlase las luces de la casa y otra que gestionase el sistema de seguridad, al utilizar la primera para encender las luces la aplicación de seguridad podría ser informada de este cambio para incluirlo en su registro.

Una vez los usuarios están unidos al bus AllJoyn permite al usuario interactuar con el mismo a través de los métodos de usuario que se hayan implementado, los cuales, a su vez, se encargarán de enviar las señales pertinentes al bus. La figura (2.9) expone gráficamente este aspecto. En P2PTrain estos métodos se encuentran en la clase AllJoynService, expuesta en el Anexo 3 de este documento.

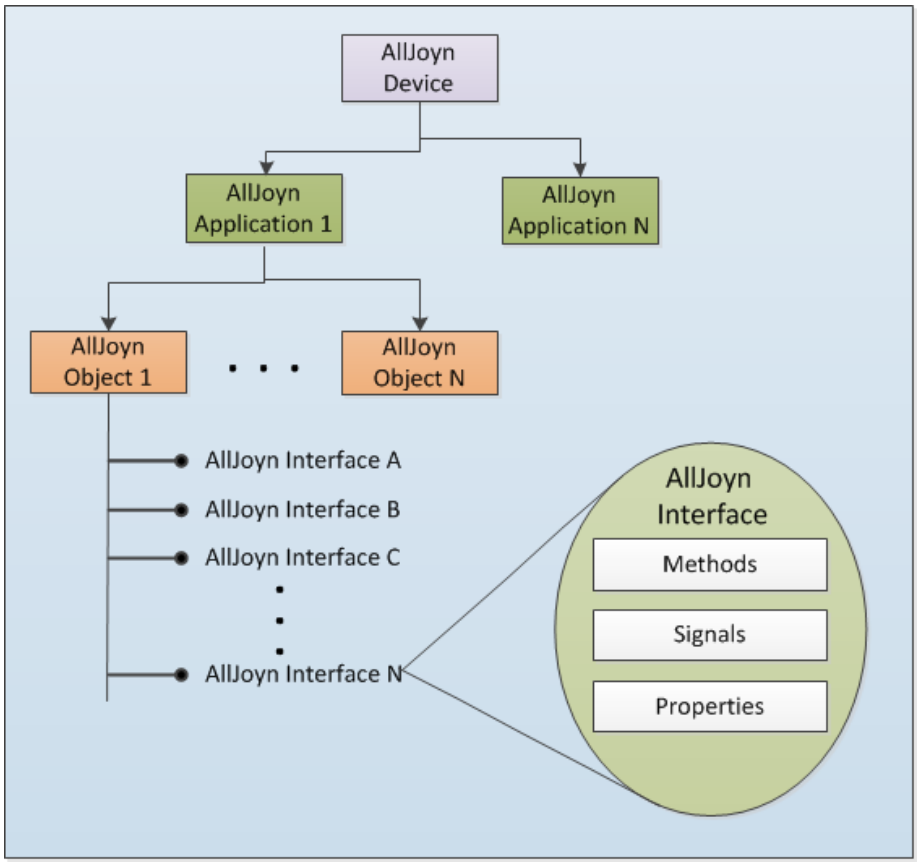

Figura 2.9 Comunicación de métodos con la interfaz de AllJoyn

Todas las funcionalidades de AllJoyn son implementadas y utilizadas en P2PTrain mediante las librerías 'alljoyn.jar' y 'liballjoyn_java.so'. La figura (2.10) muestra las librerías incluídas en la aplicación.

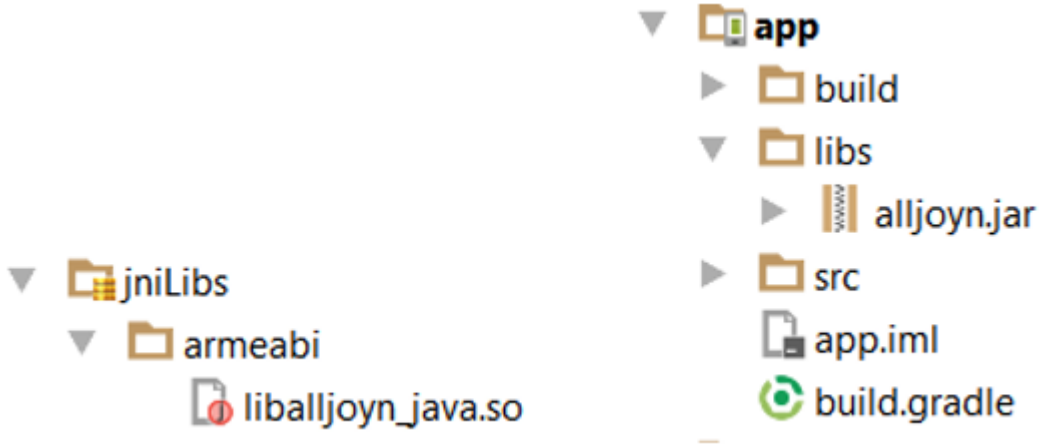

Figura 2.10 Librerías AllJoyn de la aplicación

3 INTERFAZ DE USUARIO

En este capítulo se detalla la apariencia de la interfaz gráfica de usuario que este observa en la aplicación durante su uso. El capítulo se subdivide en varios apartados, cada uno de ellos dedicado a una de las ventanas con las diversas opciones que el usuario puede llevar a cabo con la aplicación.

3.1 Introducción

La aplicación recibe el nombre P2PTrain en el dispositivo. Tras su instalación, detallada en el Anexo C, el usuario verá en su dock de aplicaciones el icono de la aplicación con el que poder acceder a la misma. El icono ha sido diseñado tomando de ejemplo la línea de diseño habitual de Android recientemente: iconos curvilíneos, redondos y con colores representativos y fácilmente identificables.

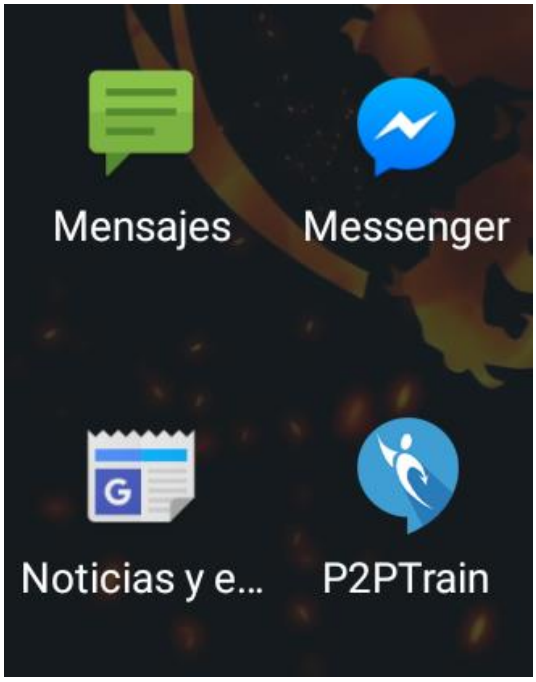

Figura 3.1 Icono de la aplicación en el dock de aplicaciones de Android

En concreto, el icono tomado de ejemplo ha sido el de la aplicación Hangouts, dada su fácil asociación con una aplicación de chat. El icono se detalla en la figura (3.2) y ha sido diseñado junto a otros recursos gráficos de la aplicación, por Iván Antonio Rodriguez, al que dedico todos mis agradecimientos.

Figura 3.2 Icono de P2PTrain

Desde el momento en el que usuario inicie la aplicación esta iniciará un servicio en segundo plano para la conexión peer-to-peer con otros usuarios. Este proceso se refleja en la parte superior de la pantalla en forma de notificación emergente.

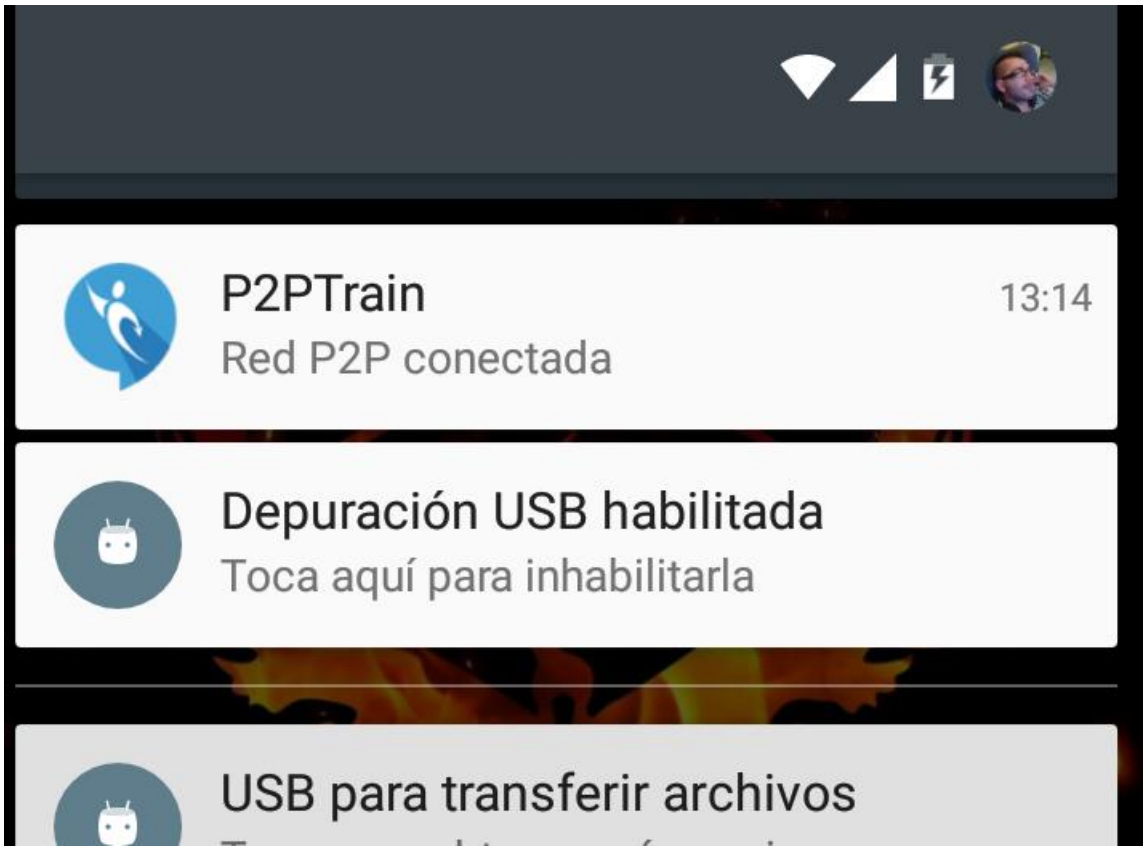

Figura 3.3 Notificación emergente de la aplicación en la interfaz de Android

3.2 Pantalla inicial

La primera ventana que observa el usuario al abrir la aplicación contiene un mensaje de bienvenida, el icono de la aplicación y un cuadro de texto en el que el usuario deberá introducir el nombre con el que desea ser identificado por otros usuarios en esta sesión de entrenamiento, tal y como se presenta en la figura (3.4). Este nombre puede variar cada vez que el usuario utiliza la aplicación y no requiere de contraseña. En el apartado 6.1 de este documento se estudia la posibilidad de cambiar este aspecto en futuras actualizaciones.

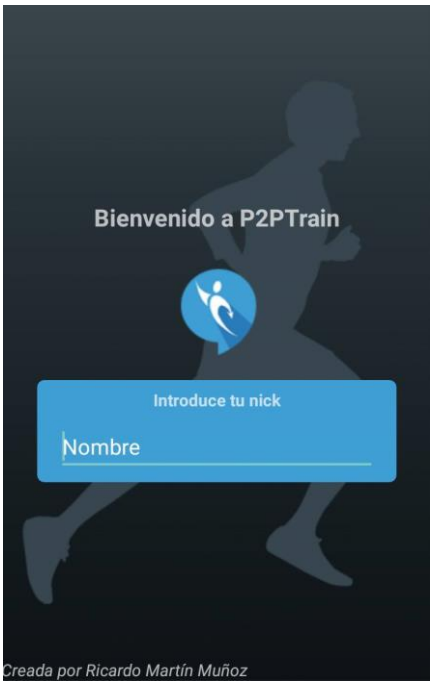

Figura 3.4 Pantalla inicial

El nombre escogido por el usuario debe tener al menos tres caracteres para ser mínimamente identificativo. De no ser así se mostrará un mensaje de advertencia invitando al usuario a introducir un nombre más largo.

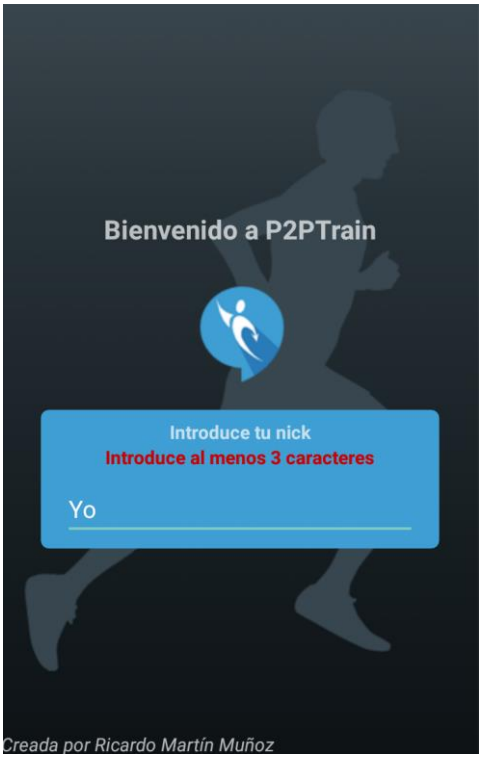

Figura 3.5 Mensaje de error por nombre demasiado corto

Una vez introducido un nombre de usuario válido, el usuario podrá acceder a la siguiente pantalla pulsando el icono central de la aplicación, que girará y emitirá un sonido antes de proceder a la siguiente actividad.

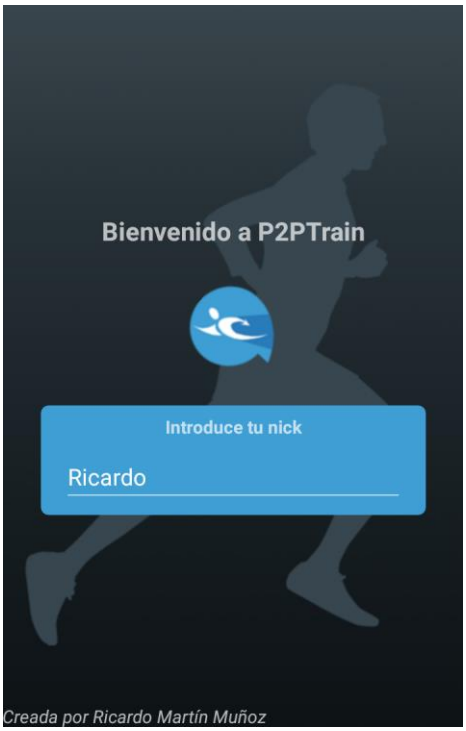

Figura 3.6 Icono girando tras introducir un nombre válido

3.3 Pantalla de chat

Inmediatamente después de la pantalla de bienvenida el usuario es llevado a la pantalla principal de la aplicación. En esta es donde el usuario podrá buscar grupos de entrenamiento y una vez se una a ellos, chatear con otros usuarios y ver y controlar el vídeo del ejercicio que proceda.

Figura 3.7 Pantalla de chat previa unión a grupo

Inicialmente la pantalla está vacía, sin vídeo ni otros usuarios con los que interactuar. El apartado en la esquina superior derecha, que muestra información del grupo al que se encuentra unido, está consiguientemente vacío, sin información que mostrar. El botón “Buscar Grupo” muestra un número que indica el número de grupos activos alrededor del usuario a los que podrá unirse. Al hacer click en este, se mostrará una ventana de diálogo

en la que elegir a cuál unirse.

Tras unirse al grupo se abrirá el fragmento de vídeo junto a los botones que controlan su reproducción. Además, se añadirá la información del grupo al cuadro superior derecho, se bloqueará el botón de "Buscar Grupo" y se desbloqueará el botón de "Dejar Grupo". De clickar en el cuadro superior derecho se mostrará un mensaje con la descripción de grupo.

Ahora el usuario puede mandar mensajes en el chat a otros usuarios en el grupo y estos a él. La comunicación es multicast, por lo que los mensajes de cada usuario son mostrados a todos por igual. Además, al controlar la reproducción del vídeo se mostrará un mensaje en el chat indicando qué usuario ha pausado o reanudado el vídeo.

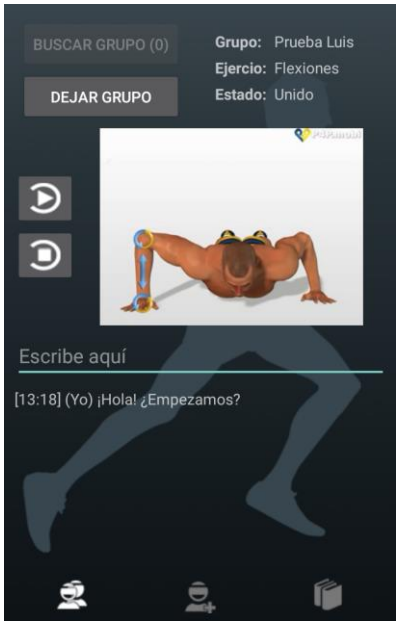

Figura 3.8 Uso del chat tras unirse a un grupo

En la parte inferior de la pantalla se muestran los botones para acceder a las dos pestañas restantes de la aplicación. El botón de la pestaña en curso se muestra en blanco y los dos restantes en gris. El primer botón abre la pestaña actual, la interfaz de chat, el segundo lleva a la pantalla de creación de grupos y el tercero al historial de ejercicios.

3.4 Pantalla de creación de grupo

A través de esta interfaz el usuario puede crear y gestionar su propio grupo de entrenamiento. Tanto si desea entrenar con más gente como si desea hacerlo solo, es en esta pantalla donde dará forma a su grupo de entrenamiento.

Figura 3.9 Pantalla de creación de grupo

La pantalla se divide en tres bloques y el botón de salida. El primero contiene la información introducida del grupo: Nombre, ejercicio, descripción y estado.

El segundo contiene los botones que permiten ajustar estos parámetros. Además contiene una 'checkbox' que permite establecer el grupo como privado, de modo que, si otros usuarios desean unirse a él, se solicitará confirmación primero al propietario. Los únicos parámetros estrictamente necesarios de establecer son el nombre del grupo y el ejercicio a realizar, de los cuales se muestra el método de introducción en las figuras (3.10) y (3.11) respectivamente: a través de un cuadro de texto en el que escribir el nombre y una lista de la que seleccionar tocando el ejercicio deseado.

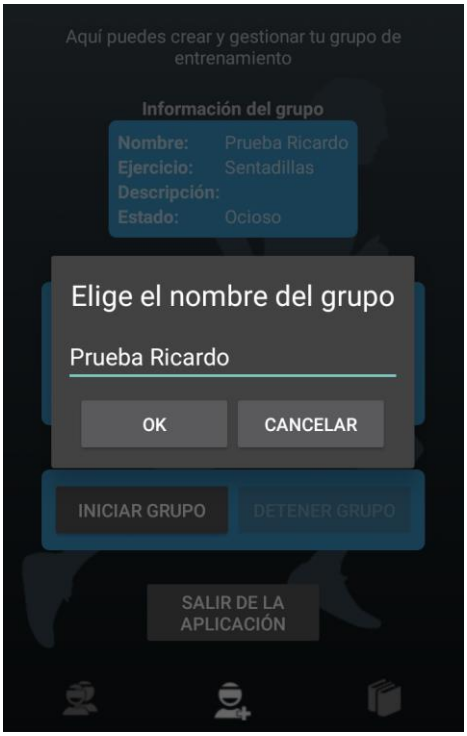

Figura 3.10 Diálogo de introducción de nombre del grupo

Figura 3.11 Selección de ejercicio

El tercer bloque contiene los botones para iniciar la difusión del grupo o detenerla. Estos botones son alternativos: si uno está activo el otro no lo está. Al iniciarse un grupo los botones del segundo bloque quedan inactivos, excepto el de selección de ejercicio, para poder ir cambiando de ejercicio según avance la rutina.

Figura 3.12 Grupo creado y botones bloqueados

Al utilizar el botón de iniciar grupo el usuario creará el grupo, que comenzará a anunciarse a otros dispositivos, y se unirá al mismo como propietario. Al detenerlo abandonará y destruirá el grupo. Finalmente, el botón en la

parte inferior de la pantalla sirve para detener el proceso de anuncio y descubrimiento p2p y salir de la aplicación. De no utilizarlo y cerrar la aplicación de manera convencional esta quedaría activa en segundo plano. Se ha hecho de este modo para que los grupos no se detengan si el usuario cierra accidentalmente la aplicación o su dispositivo ejecuta otra aplicación como una llamada entrante.

3.5 Pantalla de historial

La tercera pestaña de la aplicación contiene el historial de ejercicios del usuario. Este puede, de un rápido vistazo, comprobar las rutinas de entrenamiento que ha ido realizando o, si lo desea, borrarla por completo.

Figura 3.13 Pantalla de historial de ejercicios

El historial puede contener hasta un máximo de 50 ejercicios, a partir de los cuales comenzará a borrar los más antiguos para ir incluyendo los más recientes.

El historial de ejercicios se gestiona a través de consultas a la base de datos interna de la aplicación, en la cual se incluye una nueva entrada cada vez que el usuario comienza un ejercicio nuevo.

4 MODELADO DE LA INFORMACIÓN

Este capítulo sirve de muestra del modelado de la arquitectura que conforma la aplicación y como esta maneja la información proporcionada por el usuario y la red.

4.1 Modelo MVC

La arquitectura Modelo-Vista-Controlador (MVC) es un patrón de arquitectura de software que separa la lógica, los datos y la interfaz de usuario de una aplicación. Esta arquitectura divide los componentes de una aplicación en tres módulos, reflejados en la figura (4.1).

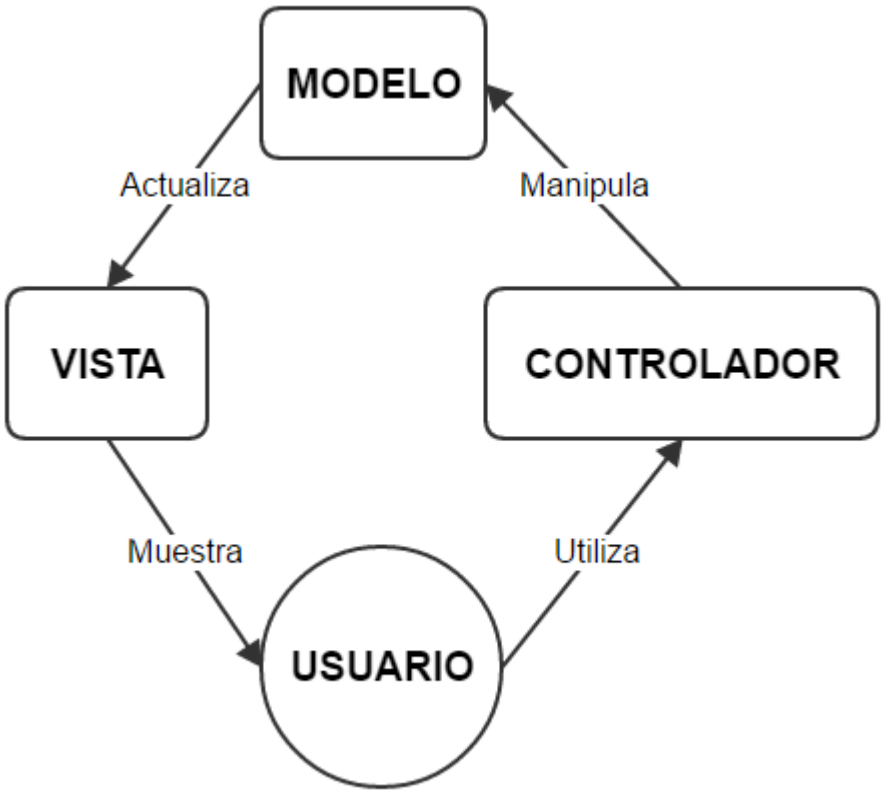

Figura 4.1 Arquitectura MVC

El Modelo contiene la información con la que opera el software y gestiona los accesos a la misma, tanto para consultas como modificaciones. La Vista es la representación por pantalla de esta información para que pueda ser percibida por el usuario. No toda la información del Modelo tiene por qué ser expuesta al usuario en la Vista, pero sí la suficiente para que pueda hacer uso de la aplicación. Finalmente el Controlador es la parte del software mediante la cual el usuario puede interactuar con el Modelo, introduciendo información en el sistema o modificándola.

P2PTrain sigue la estructura de MVC a través de la implementación de sus diversas clases.

La Vista de P2PTrain la forman las distintas pantallas de la interfaz gráfica que el usuario percibe, detalladas en el apartado 3 de este documento.

- Pantalla de inicio.
- Pantalla de grupo y chat.
- Pantalla de creación y gestión de grupos.

- Pantalla de historial.

Estas pantallas son implementadas mediante de los ficheros alojados en la carpeta de 'layout' de la aplicación, representados en la figura (4.2).

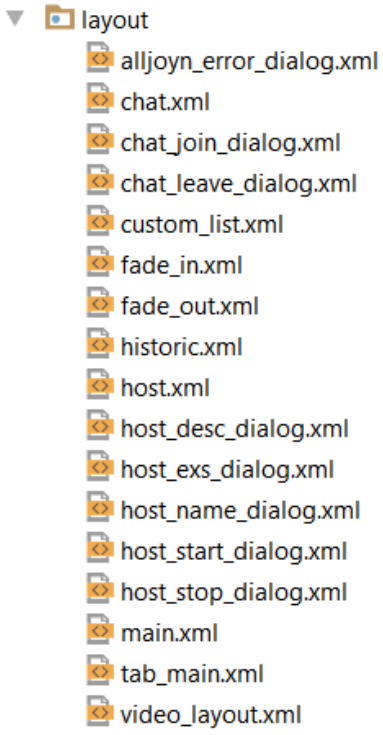

Figura 4.2 Ficheros layout de la aplicación

Estos ficheros, escritos en lenguaje xml, controlan la apariencia de las pantallas y todos los elementos emergentes de la aplicación, como cuadros de diálogo y mensajes de alerta. Además, existen otros ficheros que contienen otros aspectos gráficos, como los iconos y el fondo de la aplicación. Dichos ficheros se almacenan en la carpeta 'drawable'.

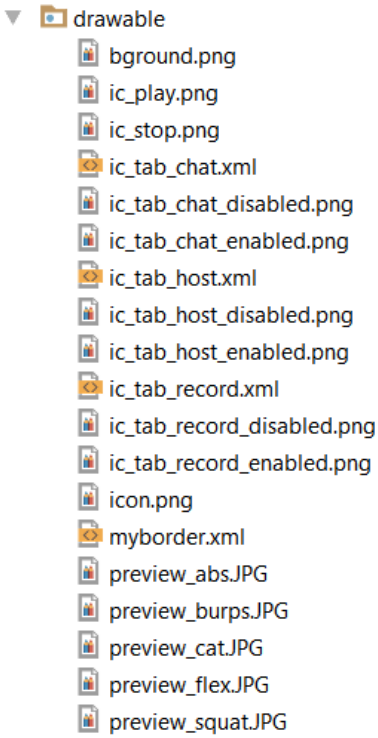

Figura 4.3 Ficheros gráficos de la aplicación

El Controlador de P2PTrain se compone de las clases que controlan los métodos que el usuario puede ver en estas pantallas y que le permiten interactuar con la aplicación. Estas clases, escritas en Java, están explicadas en el apartado 5.3 y el Anexo 3 de este documento.

Finalmente, el Modelo de P2P está conformado por dos grupos de datos:

1. La información estática alojada en la aplicación, compuesta por los vídeos de los ejercicios y los sonidos empleados, almacenados en la carpeta 'raw'.
2. La información variable de la aplicación, formada a su vez por:
    - Los datos del historial de ejercicios alojados en la base de datos.
    - Los datos de la conexión peer-to-peer de AllJoyn, transmitidos por el bus multicast.

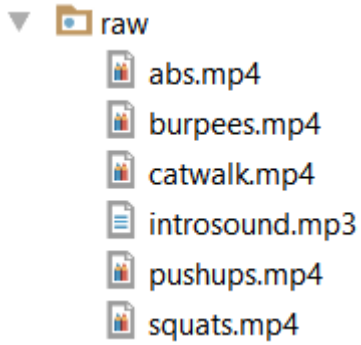

Figura 4.4 Ficheros de ejercicios y sonidos

4.2 Base de datos

P2PTrain hace uso de la tecnología SQLite para crear una base de datos local y gestionar la misma mediante consultas y modificaciones de su contenido.

Esta base de datos local recibe el nombre de P2PTrainDB y consta de una única tabla en la que se almacenan los ejercicios que el usuario va realizando para mostrarlos posteriormente en la pantalla del historial de ejercicios. Dicha tabla recibe el nombre de 'Exercises' y cuenta con cuatro columnas:

1. 'id': valor de tipo INT (Entero) utilizada como clave primaria de la tabla. No contiene ninguna información del ejercicio, es una simple variable de apoyo para mantener la tabla ordenada.
2. 'exercise': valor de tipo TEXT (texto) que contendrá el nombre del ejercicio realizado. En la versión de P2PTrain desarrollada en este trabajo se han implementado cinco ejercicios (flexiones, abdominales, sentadillas, burpees y gateadas), por lo que los posibles valores de este campo están limitados a los nombres de esos cinco ejercicios.
3. 'group': valor de tipo TEXT que contendrá el nombre del grupo en el que se ha realizado el ejercicio. Dado que los usuarios pueden introducir cualquier nombre para sus grupos, este campo puede contener cualquier valor también.
4. 'date': valor de tipo TEXT que contendrá la fecha y hora en la que se ha realizado el ejercicio. Se ha decidido implementar este valor como tipo TEXT en lugar de DATE (Fecha) por comodidad a la hora de manejar las conversiones de datos en la aplicación.

Figura 4.5 Base de datos P2PTrainDB

La figura (4.5) expone una representación gráfica de la base de datos citada.

4.3 Flujo de información

La transmisión de datos en una aplicación es un factor determinante en su comportamiento y un aspecto relevante a la hora de comprender correctamente su funcionamiento.

En P2PTrain existen dos flujos de datos: uno externo y uno interno.

1. El flujo de datos externo es el que se produce al comunicarse la aplicación con otra instancia de la misma de un dispositivo remoto. Este flujo de datos viaja sobre el bus multicast de AllJoyn encapsulado en mensajes del protocolo TCP. La figura (4.5) representa dicho flujo:

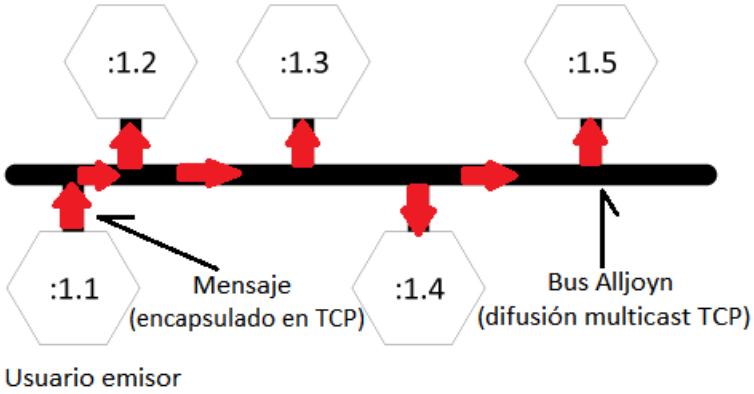

Figura 4.6 Flujo de mensajes entre usuarios remotos

2. El flujo de datos interno es el resultante de la comunicación entre la aplicación y la base de datos local en la que se almacenan los ejercicios realizados. Este tráfico nunca genera mensajes que se envíen fuera del dispositivo. La comunicación se lleva a cabo a través de la clase DBHandler implementada con SQLite, detallada en el Anexo 3 de este documento.

5 ANÁLISIS DE LA APLICACIÓN

En este capítulo se exponen los diferentes diagramas que detallan el proceso interno de funcionamiento de la aplicación en distintos ámbitos. Dichos diagramas se exponen en UML (Unified Modeling Language, o Lenguaje de Modelado Unificado en español).

Los diagramas expuestos son los de casos de uso, de secuencia y de clases.

5.1 Diagramas de casos de uso

Detallan los diversos casos en los que la aplicación puede ser usada y quien realiza cada uno de estos usos.

5.1.1 Identificación de actores

Usuario local: es el actor que hace uso de la aplicación en su dispositivo y el único estrictamente necesario. Puede acceder a todas las funciones del sistema.

Usuario remoto: es el actor que hace uso de la aplicación en otro dispositivo que conectará con el del usuario local. Solo tiene acceso a algunas funciones de la aplicación local.

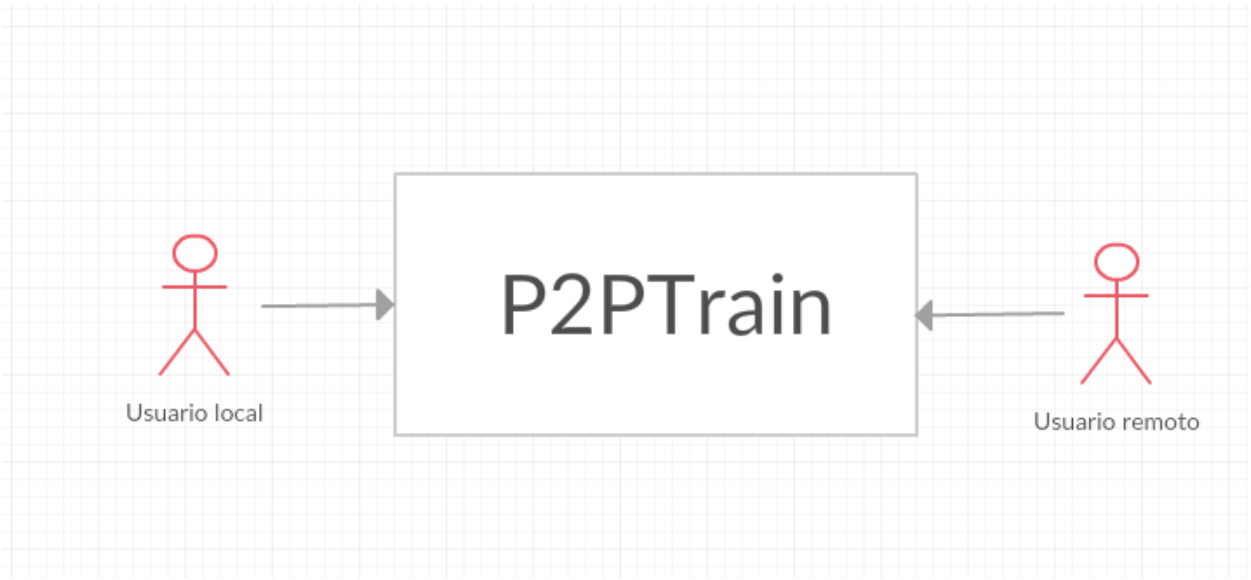

Figura 5.1 Identificación de actores

5.1.2 Casos de uso

La figura (5.2) detalla los casos de uso de la aplicación. Cada uno de estos casos de uso se destaca a continuación en un subapartado propio que explica su comportamiento individual.

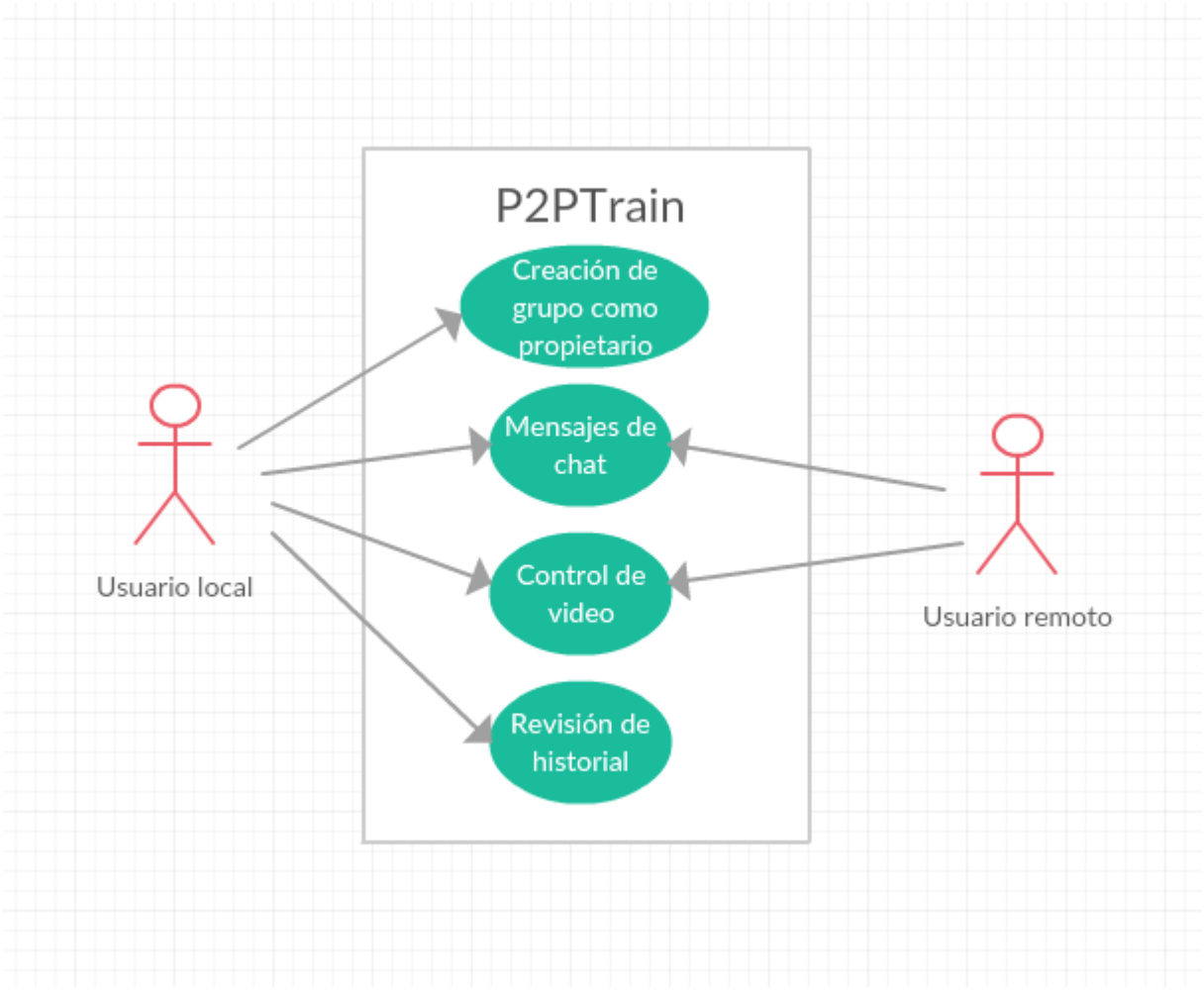

Figura 5.2 Casos de uso

5.1.2.1 Creación de grupo como propietario

Corresponde al proceso de creación de un grupo de entrenamiento y la consiguiente incorporación del usuario al mismo como propietario. Este caso de uso es exclusivo del usuario local ya que la aplicación no nos permite crear un grupo médiate un dispositivo remoto, solo desde el propio usuario.

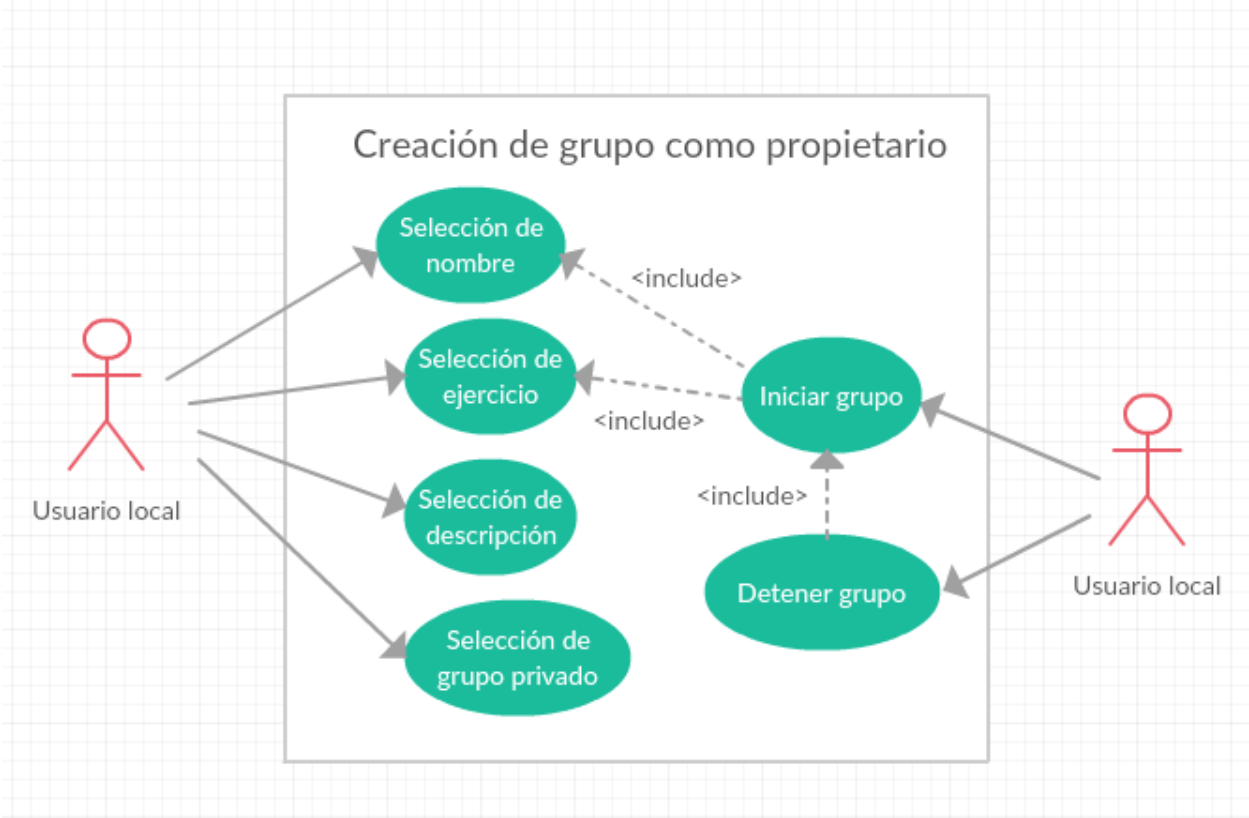

Figura 5.3 Caso de uso I – Creación de grupo

Para crear un grupo el usuario debe, forzosamente, establecer primero el nombre y el ejercicio del mismo. Una vez que lo ha hecho puede iniciar el grupo.

Para poder detener el grupo, evidentemente, el usuario ha debido iniciarlo primero.

5.1.2.2 Mensajes de chat

El uso del servicio de chat es una funcionalidad habilitada para los dos extremos de la comunicación peer-to-peer en la aplicación. Para enviar mensajes es necesario que el usuario esté unido a un grupo previamente, aunque este puede escribir mensajes sin estar unido a un grupo. Estos mensajes sin embargo no serán mandados si el usuario trata de enviarlos, solo se expondrán de manera local, pero al no haber ninguna conexión peer-to-peer el mensaje no será enviado por ninguna interfaz.

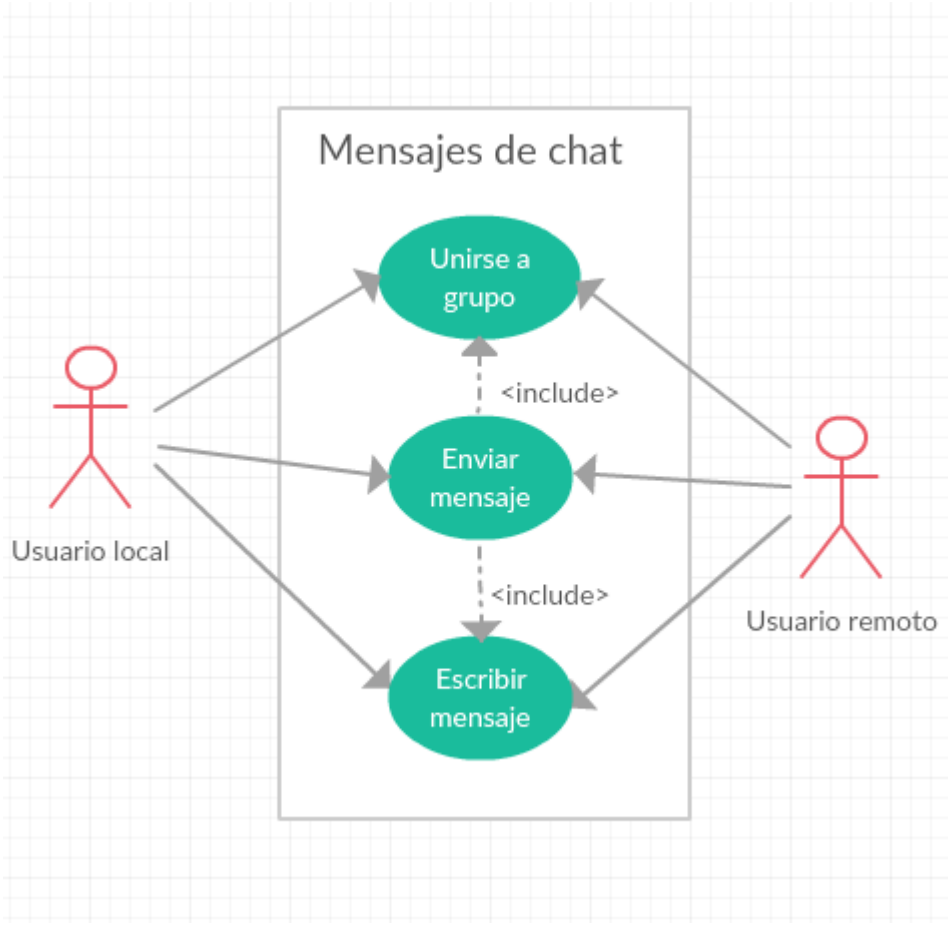

Figura 5.4 Caso de uso II – Mensajes de chat

Los mensajes del usuario remoto son mostrados en la aplicación local mediante la interfaz local y para ello el usuario remoto ha debido seguir el mismo proceso: incorporación al grupo, introducción y envío de mensaje.

5.1.2.3 Control de vídeo

Del mismo modo que los mensajes de chat, para controlar la reproducción del vídeo explicativo del ejercicio el usuario debe primero estar unido al grupo correspondiente. Por defecto, cuando se inicia un grupo el vídeo se encuentra en pausa.

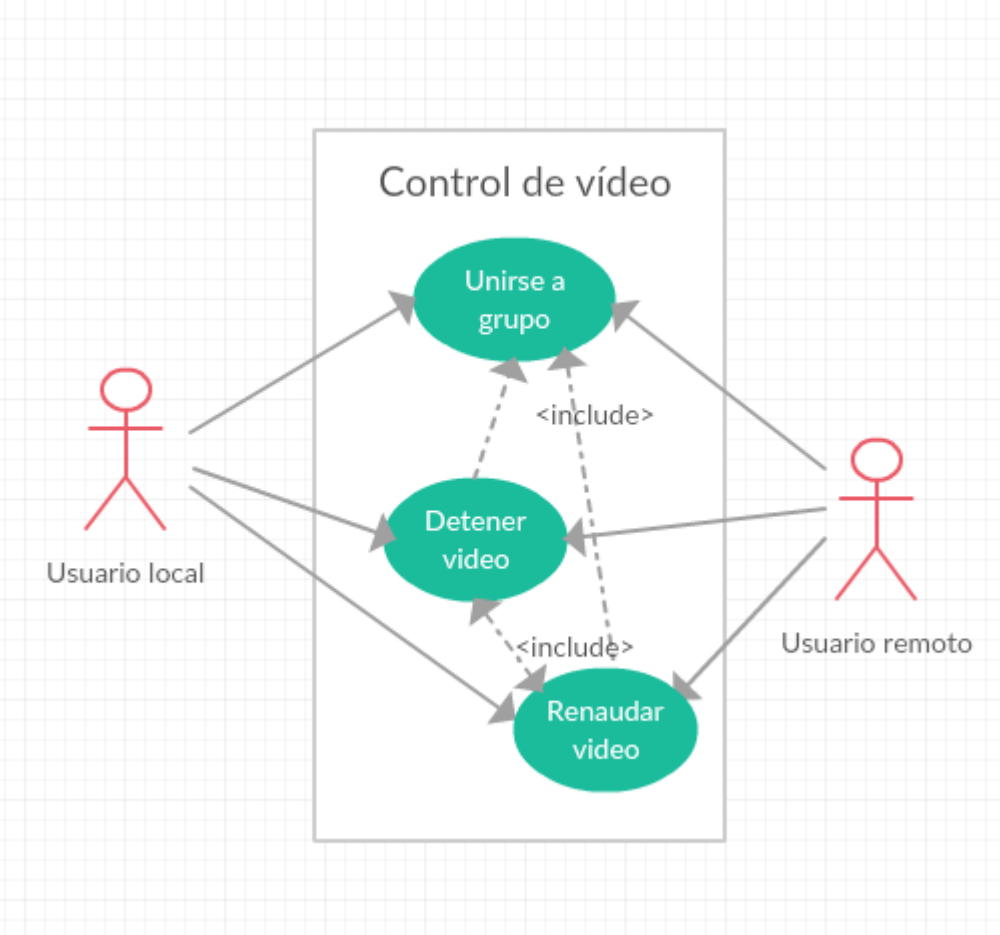

Figura 5.5 Caso de uso III – Control de vídeo

5.1.2.4 Revisión del historial

La función de historial de ejercicios es exclusivamente local, lo que significa que no es posible consultar el historial de ejercicios desde un dispositivo remoto. Para hacer uso de ella el usuario debe, al menos, haber participado en un ejercicio, lo cual significa haberse unido a un grupo, al menos una vez.

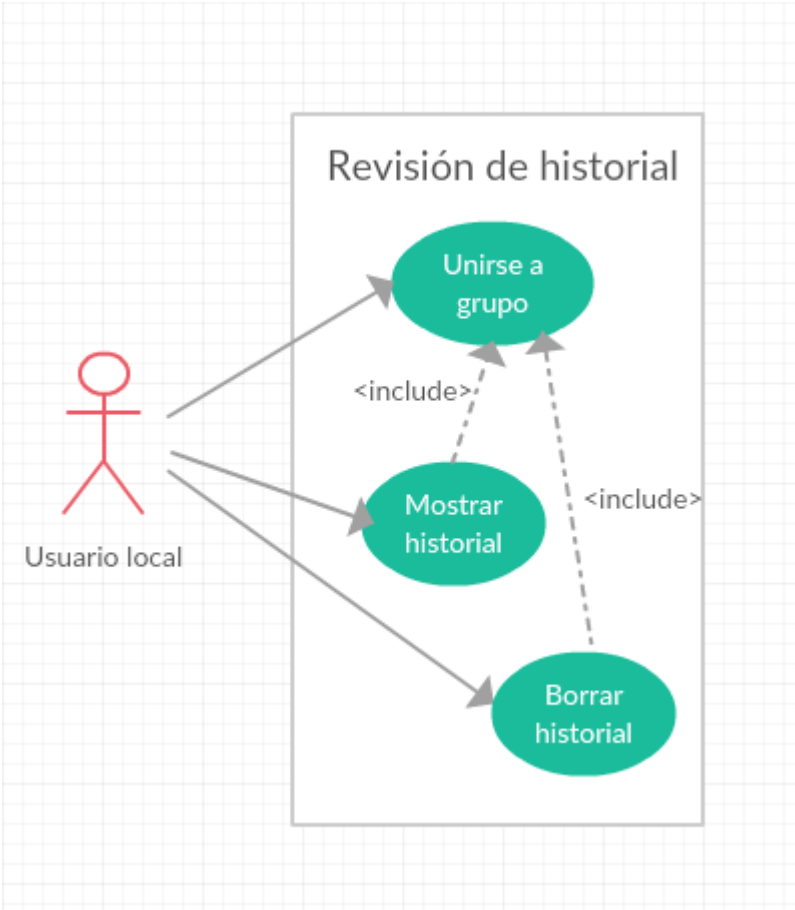

Figura 5.6 Caso de uso IV – Revisión de historial

5.2 Diagramas de secuencia

Los diagramas de secuencia representan el intercambio de mensajes entre los distintos componentes del escenario ante determinados eventos. El eje vertical del diagrama representa el tiempo, mientras que el horizontal plasma los distintos espacios de comunicación entre los componentes.

5.2.1.1 Creación de grupo público e incorporación

Es el proceso mediante el cual el usuario local crea un grupo de carácter público, sin restricciones de acceso, y un usuario remoto se incorpora al mismo.

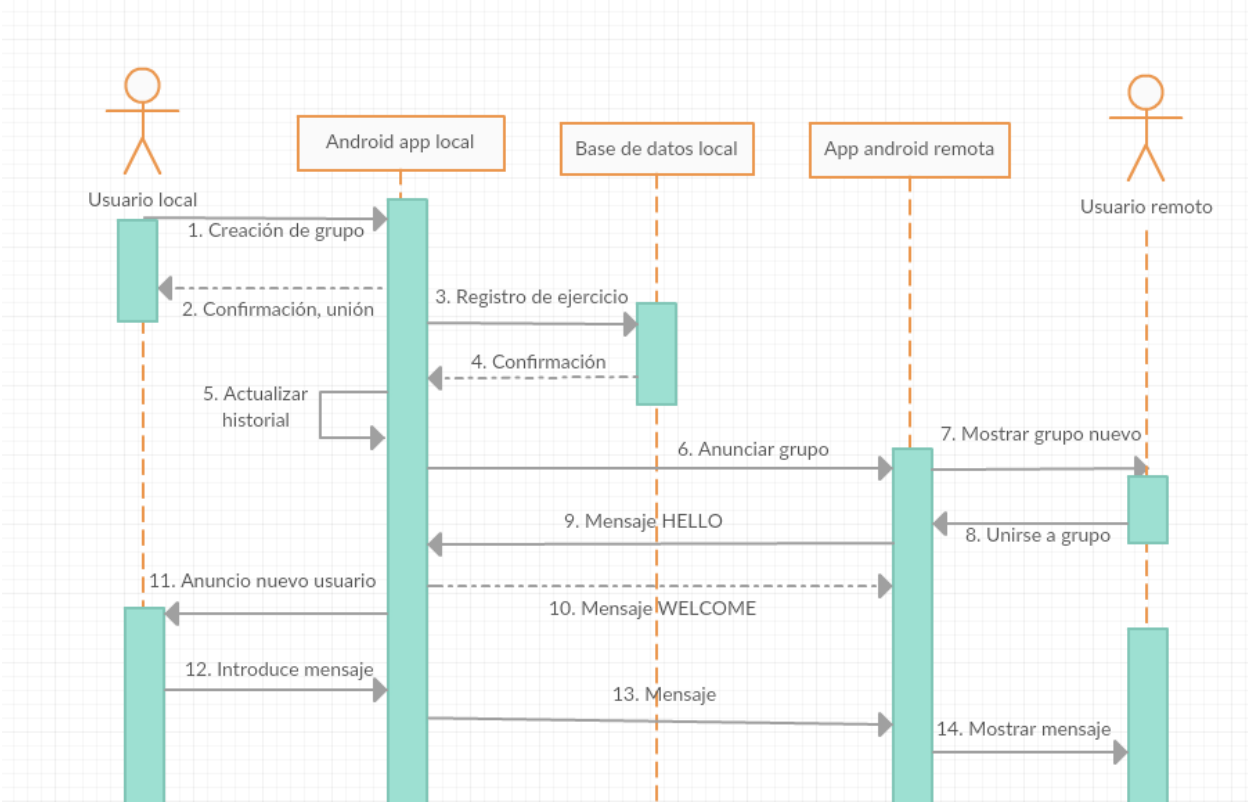

Figura 5.7 Creación de grupo público e incorporación de usuario

Los mensajes HELLO y WELCOME contienen información acerca del usuario remoto y el grupo de entrenamiento, respectivamente, tales como el nombre del primero y el ejercicio y estado del mismo en el segundo.

5.2.1.2 Creación de grupo privado e incorporación aceptada y rechazada

Al crear un grupo privado el usuario gana el privilegio de decidir si desea aceptar a los usuarios que tratan de incorporarse al mismo o si, por el contrario, los rechaza. Esta decisión es tomada a través de un sencillo mensaje emergente en la pantalla del propietario.

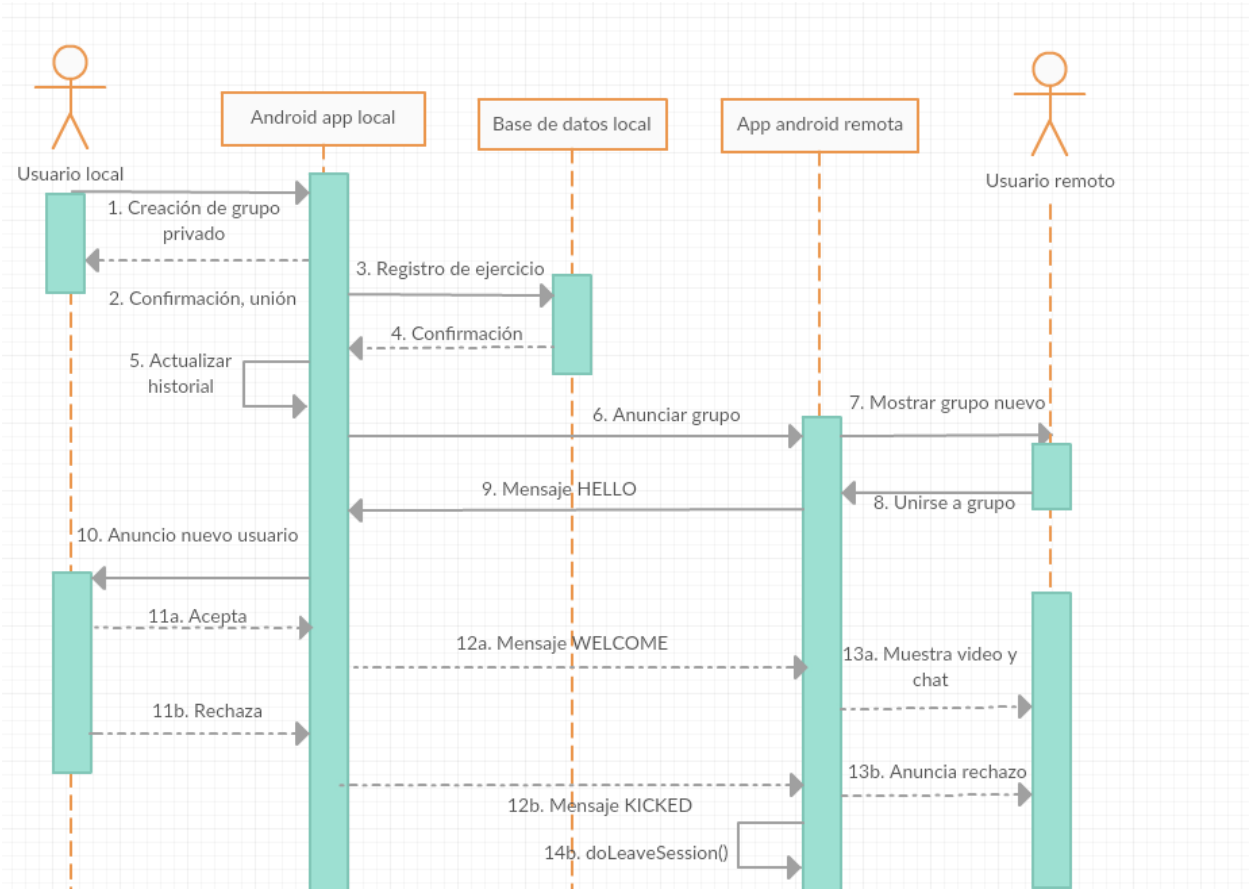

Figura 5.8 Creación de grupo privado e incorporación de usuario con y sin rechazo

El mensaje KICKED informa a todos los usuarios remotos de que el usuario que ha tratado de unirse ha sido rechazado. Este abandona el grupo y cierra la interfaz de vídeo y chat.

5.2.1.3 Cambio de ejercicio

Durante la realización de las rutinas será frecuente el cambio de ejercicio para entrenar diversos grupos musculares. La figura (5.9) refleja el intercambio de mensajes que tiene lugar en ese caso.

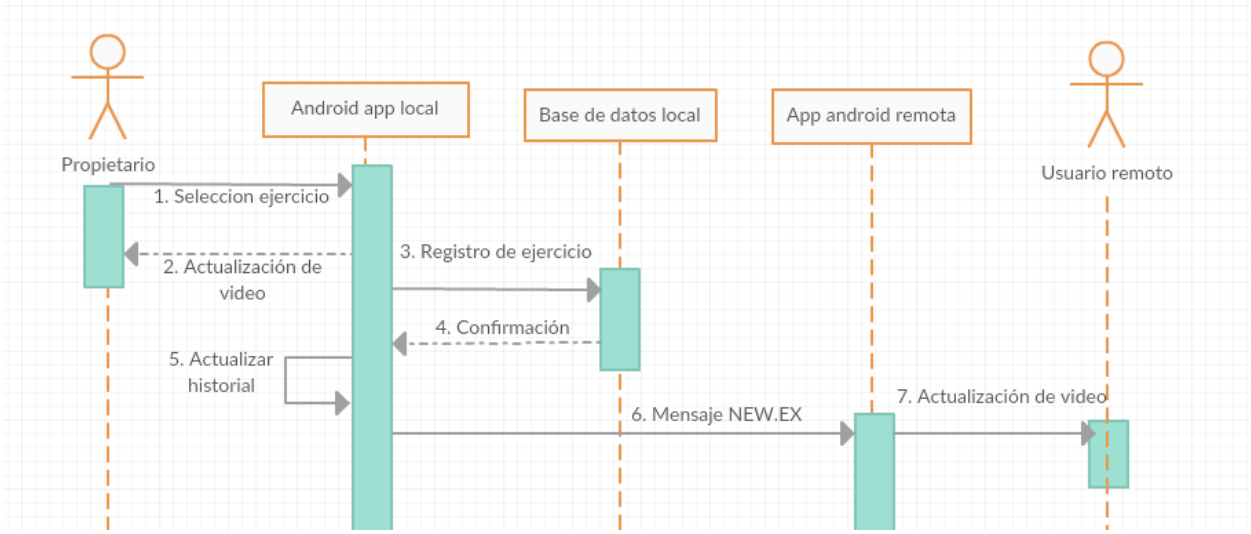

Figura 5.9 Cambio de ejercicio

El mensaje NEW.EX informa a los usuarios remotos de que se ha cambiado de ejercicio y cuál es el que se

llevará a cabo ahora.

5.2.1.4 Control de reproducción de video

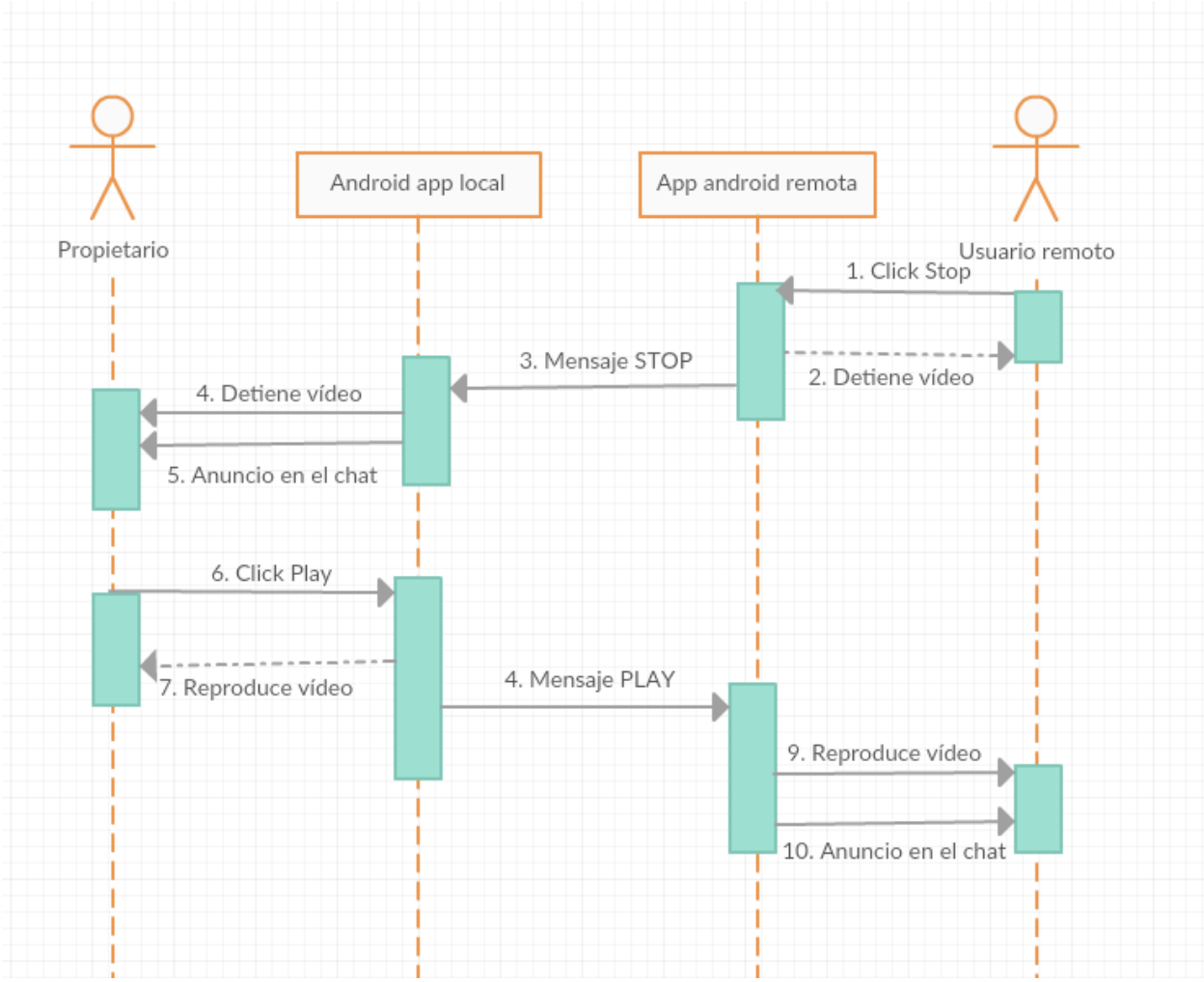

Figura 5.10 Control de reproducción de vídeo de ejercicio

La figura (5.10) detalla el intercambio de mensajes que sucede al detener o reanudar el vídeo por parte de los usuarios. Los mensajes STOP y PLAY indican a los usuarios remotos que el vídeo se detiene o se vuelve a reproducir. Además, contienen información para sincronizar el vídeo en caso de deslizamiento de tramas, incluyendo el segundo actual en el que debe encontrarse el vídeo.

Junto con el control del vídeo, la aplicación envía mensajes de anuncio en el chat para reflejar qué usuario es responsable del cambio de estado en el vídeo.

5.2.1.5 Abandono y/o cierre de un grupo

El proceso de abandono de un grupo puede darse de dos modos. De manera directa, al usar el botón correspondiente en la pantalla de chat de la interfaz de usuario, o bien de manera indirecta al cerrar la aplicación con el botón de la pantalla de creación de grupos o si se es propietario, con el botón de “Detener grupo”.

En el caso de ser un usuario remoto, en ambos casos se envía en un mensaje BYE a la red P2P, pero en el caso de salir sin cerrar la aplicación esta se mantendrá en la pantalla de chat, sin vídeo, a la espera de que el usuario se una a otro grupo o cree uno él mismo.

En el caso de ser el propietario del grupo se envía un mensaje KILL a todos los miembros del grupo, forzándoles así a abandonar el grupo. Tras ello se cancela el anuncio del grupo en la red P2P y se sale del mismo, liberando el nombre y los recursos asociados en el proceso.

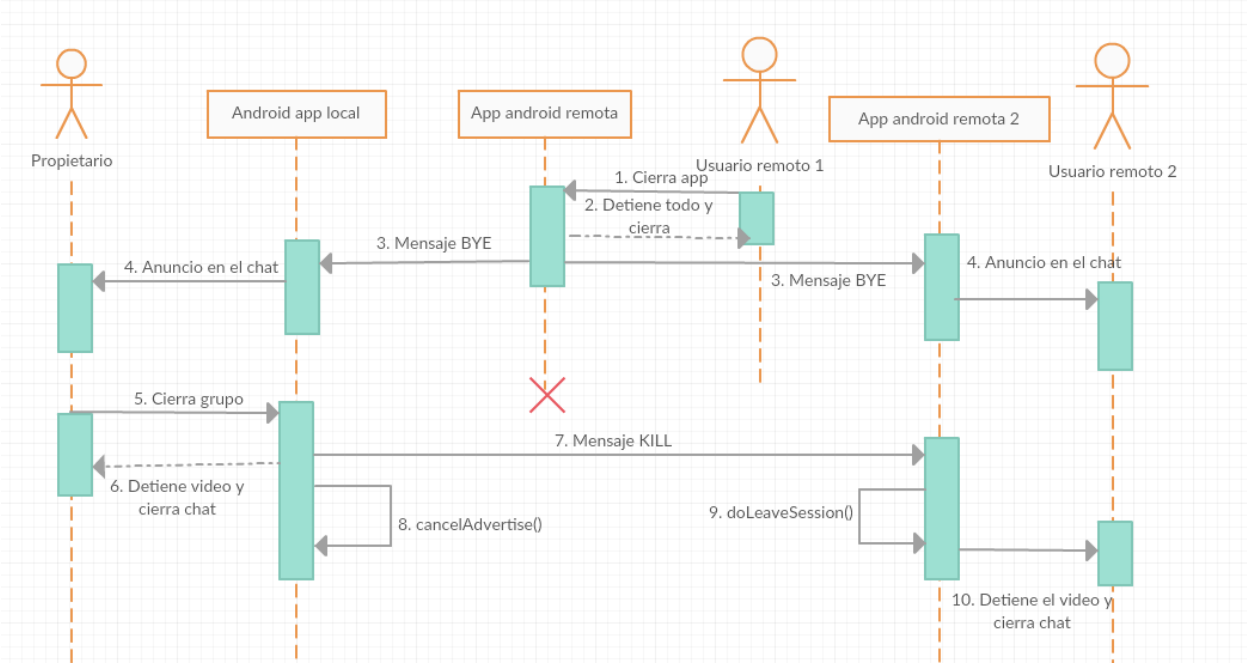

Figura 5.11 Abandono y cierre de grupo

5.3 Diagramas de clases

El diagrama de clases representado en la figura (5.12) refleja la estructura de clases Java de la aplicación, sus funciones y las relaciones de uso entre ellas.

En total P2PTrain cuenta con trece clases y tres interfaces distintas, cada una de las cuales encargada de desempeñar una función concreta en la aplicación. Las clases más relevantes por su uso más repetido y abundancia de código son AllJoynService, TrainApplication, HostActivity y ChatActivity.

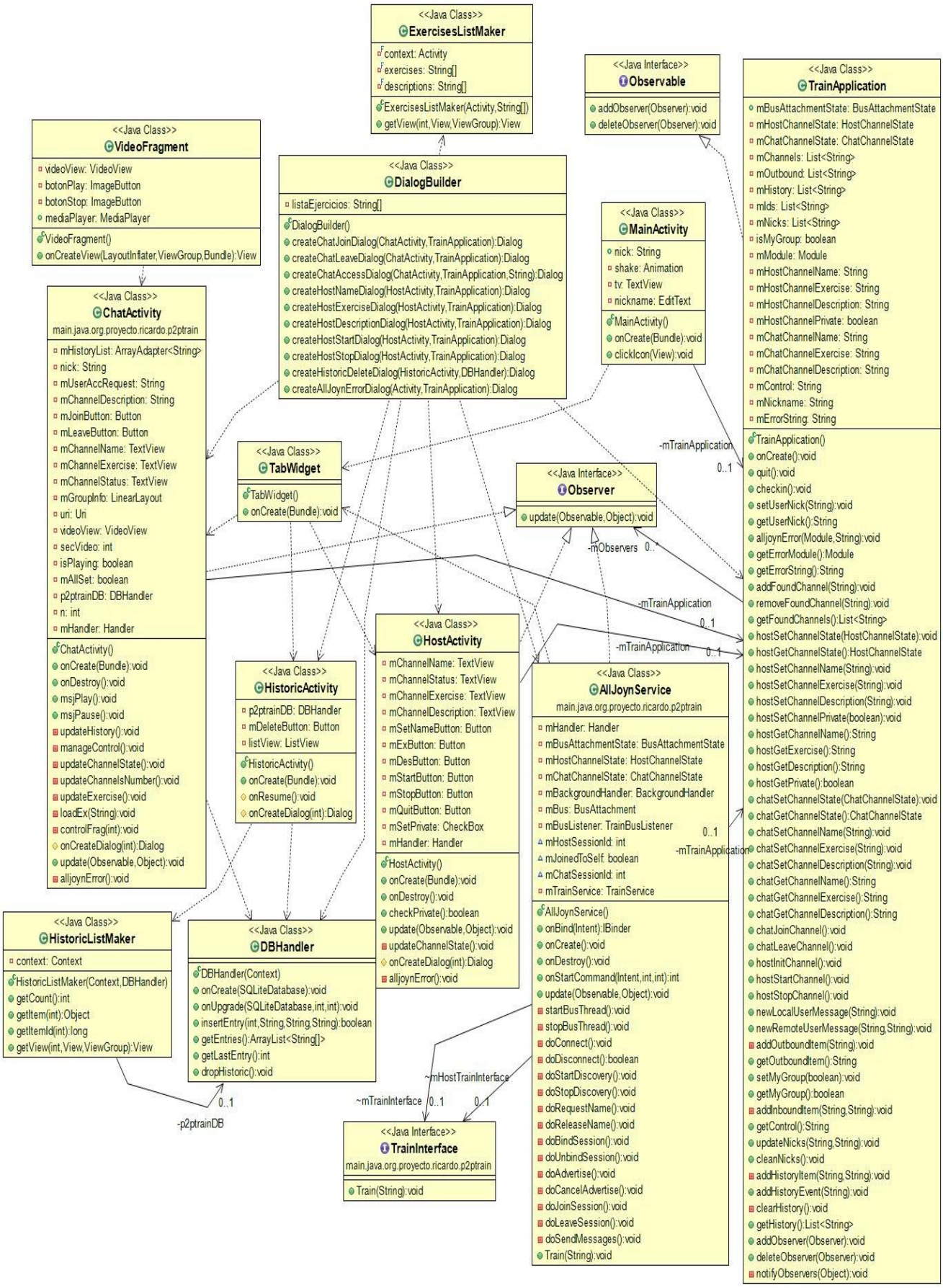

Figura 5.12 Diagrama de clases

P2PTrain cuenta con las siguientes clases:

- AllJoynService: es la clase que interactúa con las librerías de AllJoyn importadas en el fichero alljoyn.jar. La mayoría de sus métodos son estándares extraídos directamente de la documentación de AllJoyn. La clase TrainApplication hace uso de ella.

  Los métodos 'do' (doConnect(), doRequestName(), doSendMessages(), etc) son los que acceden directamente a las librerías de AllJoyn para realizar todas las funciones relativas al intercambio de información peer-to-peer.

  Por ejemplo doConnect() es empleada para unirse a al bus AllJoyn. Para ello hace uso de un objeto de tipo BusAttachment en el cual registra un listener que se mantendrá a la escucha de mensajes procedentes del bus. El método doAdvertise() se encarga, por otro lado, de anunciar el nombre de los grupos creados por el usuario haciendo uso de la función advertiseName() propia de los objetos de tipo BusAttachment.

  Todos estos métodos actúan de forma similar, acudiendo a funciones de la librería de AllJoyn de manera que el desarrollador no necesita implementar los medios de comunicación de la red P2P por si mismo.

- ChatActivity: es la clase encargada de la pantalla de chat de la interfaz de usuario. Utiliza el layout chat.xml para dar forma a la misma y se encarga, además, de gestionar los mensajes de control como NEW.EX, WELCOME, HELLO…

- DBHandler: clase encargada de realizar todas las gestiones con la base de datos interna de la aplicación. Las clases ChatActivity e HistoricActivity hacen uso de ella.

- DialogBuilder: clase encargada de crear todos los diálogos que se muestran por pantalla cuando se le ofrecen diversas opciones al usuario (unirse a grupo, introducir el nombre del grupo a crear, seleccionar el ejercicio…).

- ExercisesListMaker: clase tipo Adapter utilizada para crear la lista personalizada que da forma a la interfaz de selección de ejercicio al crear grupo.

- HistoricActivity: es la clase encargada de la pantalla de historial de la interfaz de usuario. Utiliza el layout historic.xml para dar forma a la misma y se apoya en DBHandler para obtener la lista de ejercicios del historial.

- HistoricListMaker: clase tipo Adapter utilizada para crear la lista personalizada que da forma al historial de ejercicios en la actividad HistoricActivity.

- HostActivity: es la clase encargada de la pantalla de creación de grupos de la interfaz de usuario. Utiliza el layout host.xml para dar forma a la misma. Es desde esta clase desde la que se puede cerrar por completo la aplicación en lugar de dejarla en segundo plano.

- MainActivity: es la clase principal que se encarga de la pantalla inicial de la interfaz de usuario. Contiene métodos para comprobar la validez del nick elegido, pasarlo como parámetro al resto de la aplicación y añadir efectos audiovisuales a la transición al resto de la aplicación.

- Observable: es una interfaz para facilitar la comunicación entre TrainApplication y las clases que implementan la interfaz Observer.

- Observer: es la interfaz que usada por las clases ChatApplication y HostApplication para comunicarse con TrainApplication. Esta utiliza una interfaz de objeto observable que envía notificiaciones a los objetos observadores cuando se producen distintos eventos.

- TabWiget: es la clase que genera las pestañas que conforman las tres pantallas de la interfaz de usuario. Al implementarlo de este modo el usuario puede navegar entre las tres pantallas sin que se interrumpa el contenido de las mismas.

- TrainApplication: clase mediadora entre la parte de usuario reflejada en las clases HistoricActivity y ChatActivity y las funcionalidades de AllJoyn incluidas en AllJoynService. De este modo todas las clases que hacen uso de dichas funcionalidades acceden a ellas a través de métodos comunes.

- TrainInterface: es la interfaz que usa AllJoynService para comunicarse con TrainApplication. Tomada de la documentación de AllJoyn.
- VideoFragment: clase responsable de la visibilidad del fragmento de vídeo del ejercicio y los botones asociados a este.

6 CONCLUSIONES

En este último capítulo se exponen los posibles puntos a mejorar o desarrollar del trabajo así como las conclusiones extraídas tras la realización del mismo.

Realizar este trabajo ha supuesto un proceso no solo de desarrollo si no también de aprendizaje, a través del cual se han ido afrontando una serie de obstáculos que superar hasta la finalización del mismo. Entre dichas etapas o proceso de creación se encontraron:

- Planteamiento del problema a resolver.
- Elección de una solución efectiva para dicho problema.
- Diferenciación de la solución planteada de otras alternativas presentes en el mercado.
- Elección de las tecnologías a usar para alcanzar la solución elegida.
- Comprensión, manejo e implementación de las tecnologías elegidas.
- Codificación de la solución y programación de la misma.
- Adaptación gráfica del software. Desarrollo de medios audiovisuales para hacerlo atractivo.
- Revisión global del alcance de los objetivos.

Todos estos puntos han sido resueltos de manera parcialmente secuencial, volviendo a varios de ellos en repetidas ocasiones para optimizar los resultados obtenidos.

6.1 Posibles mejoras y líneas de desarrollo futuro

P2PTrain ofrece diversas líneas de mejora que podrían ser implementadas en función de la orientación que quisiera dársele a la aplicación. A la hora de de estudiarlas cabe destacar que si no se han implementado durante el desarrollo de este trabajo es debido al deseo de mantener la aplicación bajo dos parámetros principales:

1. Conservar una aplicación de entrenamiento para todos los niveles de usuario.
2. Mantener en todo momento el carácter peer-to-peer de la aplicación para hacerla totalmente autónoma.

Una vez observados estos puntos, varios cauces posibles de desarrollo serían:

- Dotar a los usuarios de la facultad de subir sus propios vídeos de entrenamiento y compartirlos con otros usuarios en lugar de aquellos incluidos en la aplicación. Se ha decidido, finalmente, no empelar esta funcionalidad debido al carácter de aplicación de entrenamiento estricta de la aplicación. Dado que los usuarios están, en teoría, utilizando la aplicación para mejorar su entrenamiento y que, por tanto, no son profesionales, resulta contraproducente ofrecerles la posibilidad de compartir su ejercicio, que puede estar realizado de manera incorrecta.
- Crear una funcionalidad de “usuarios profesionales” a través de la cual se reconozca a usuarios de mayor nivel y se les brinde la posibilidad de utilizar sus propios vídeos de ejercitación como se especifica en el apartado anterior. Esta idea se descartó por establecer una diferenciación entre los usuarios y por la necesidad, nuevamente, de establecer un lado servidor que coarte el carácter estrictamente P2P de la aplicación.
- Incluir una mayor cantidad de ejercicios en la lista. Esta funcionalidad es estrictamente necesaria y el único motivo por el que no se ha llevado a cabo es el carácter puramente académico de este trabajo,

que pretende servir de muestra de la aplicación, no de producto final de la misma.

- Incluir otros métodos de comunicación entre los usuarios, como conversaciones privadas entre dos usuarios fuera de un grupo. Sin embargo, esto se aleja de la idea original de la aplicación de ser de carácter social y grupal, por lo que se ha desestimado.
- Implementación de un servidor con una base de datos que controlase el acceso de los usuarios mediante una dupla usuario-contraseña. Se ha decidido no emplear esta funcionalidad para dotar al usuario final de completa libertad a la hora de utilizar la aplicación. Sin la presencia de un servidor se suprimen los tiempos de espera por consultas y se elimina la posibilidad de que la aplicación se bloquee por una caída del servidor.
- Incluir nuevas funcionalidades en los grupos, como la compartición de música que ambiente el entrenamiento. El motivo de la no inclusión de esta funcionalidad es que los propios ejercicios incluyen cierta música para dotar de ritmo el entrenamiento y además es necesario mantenerla bajo un mínimo para poder escuchar las explicaciones que se exponen para la realización de los ejercicios. Sería interesante, sin embargo, incluir la posibilidad del usuario propietario de eliminar personas de su grupo o de cederle la propiedad del grupo a otros usuarios.

6.2 Conclusiones

Tras todo lo expuesto en el presente documento y considerando el trabajo realizado, pueden plantearse una serie de conclusiones finales que sirven como rúbrica a este Trabajo Fin de Grado.

Se presentó un escenario concreto con una serie de condiciones específicas que resolver:

1. Crear una aplicación Android relativa al ejercicio y/o entrenamiento.
2. Dotarla de un carácter social a través de la creación de grupos con tecnología P2P.
3. Mantener un nivel aceptable de sencillez y atractivo en la interfaz de usuario

La aplicación creada en este trabajo cumple con estos objetivos marcados, creando un software para telefonía móvil y dispositivos similares centrado en el ámbito del entrenamiento físico personal.

P2PTrain cumple, gracias a las librerías de AllJoyn, con el estándar peer-to-peer, estableciendo una dirección multicast a la que todos los usuarios de la aplicación vierten sus mensajes para comunicarse entre sí. Además, cuentan con diversos vídeos explicativos que actúan de guía para la realización de ejercicios con los que mantenerse en buena forma física, y pueden acceder a todas las funciones de la aplicación de una manera sencilla e intuitiva.

Como valoración personal me gustaría resaltar lo enriquecedor y formativo que resulta llevar a cabo un trabajo de esta índole, así como la gran cantidad de esfuerzo necesaria para ello.

He puesto en práctica una gran cantidad de conocimientos adquiridos durante el transcurso de mi formación universitaria, como diseño modular de software, lenguaje Java, programación en Android, protocolos de telecomunicaciones y estructuras de bases de datos.

Además he adquirido nuevos conocimientos teóricos y prácticos, como la historia y distintas arquitecturas de la tecnología P2P, el uso de librerías nuevas y desconocidas y su implementación y la gestión de un gran volumen de trabajo y sus diversos enfoques. También me ha dado mucha experiencia general en el campo del desarrollo de aplicaciones móviles, lo cual resulta muy conveniente debido al constante crecimiento de este mercado.

Concluyo este trabajo dando de nuevo las gracias a todas las personas que lo han hecho posible y citando una de las frases que más me inspira a la hora de acometer proyectos como este trabajo:

"La función de un buen software es hacer que lo complejo aparente ser simple"

--Grady Booch, desarrollador de UML

Anexo 1: ENTORNO P2P

Este anexo describe superficialmente las cualidades y la historia de la tecnología P2P dado su importancia en este trabajo. La información presente en este anexo ha sido recabada del libro 'Distributed Systems Concepts and Design'[9].

A.1 Introducción

Desde la creación de Internet se comenzó a percibir que la demanda de servicios a través de la red es una variable que va a estar siempre en aumento, escalando con la población humana con acceso a la red y la variedad de aplicaciones que se le ofrece. Esto supone un aumento de la infraestructura necesaria para soportar dichos servicios, lo que conlleva un aumento en los gastos necesarios para implementar, gestionar y mantener un sistema.

La arquitectura peer-to-peer (literalmente par-a-par o entre-pares), abreviada como P2P, nació con el objetivo de eliminar esta creciente necesidad del aumento de infraestructura desplazando las necesidades operativas de los servicios a los dispositivos finales. De este modo se puede prescindir de servidores de almacenamiento de datos al ser los equipos usuarios los que contienen los datos que transmiten entre sí, y tampoco es necesario un servidor gestionando peticiones cuando los usuarios son capaces de enrutar la comunicación por sí mismos.

Se establece pues una arquitectura de software con cualidades como:

- Un diseño que hace a los usuarios contribuir con sus propios recursos al sistema.
- Una red en la que todos los nodos tienen las mismas capacidades y funcionalidades (de ahí el nombre par, ya que todos son iguales).
- La funcionalidad del sistema no depende de un sistema central.
- Los pares pueden ser anónimos al ver únicamente la red P2P en su conjunto, no a los otros pares individualmente.

A la hora de implementar estas cualidades se deben resolver distintos aspectos como el método de descubrimiento de pares o la distribución de la información en la red. Con el objetivo de satisfacer dichas necesidades surgieron los primeros 'peer-to-peer middleware', programas de tecnología P2P para el usuario. Los siguientes apartados detallan el proceso de desarrollo y los distintos tipos de este software.

A.2 Origen

La primera aplicación con la necesidad de escalabilidad a nivel global del almacenamiento y compartición de la información fue Napster en 1999.

[9] Dystributed Systems Concept and Design, 5ª edición, por George Colouris, Jean Dollimore, Tim Kindberg y Gordon Blair

Ilustración 1 Logo de Napster

Napster era un software que permitía a sus usuarios compartir ficheros de música a través de un sistema P2P a través de índices centralizados. Existía un servidor central que contenía los índices o referencias que indicaban donde se encontraban los ficheros, pero eran los usuarios los que aportaban estos ficheros, nunca el propio sistema. Este solo indicaba que usuarios se encontraban en posesión de los ficheros para que otros pudieran descargarlos de ellos.

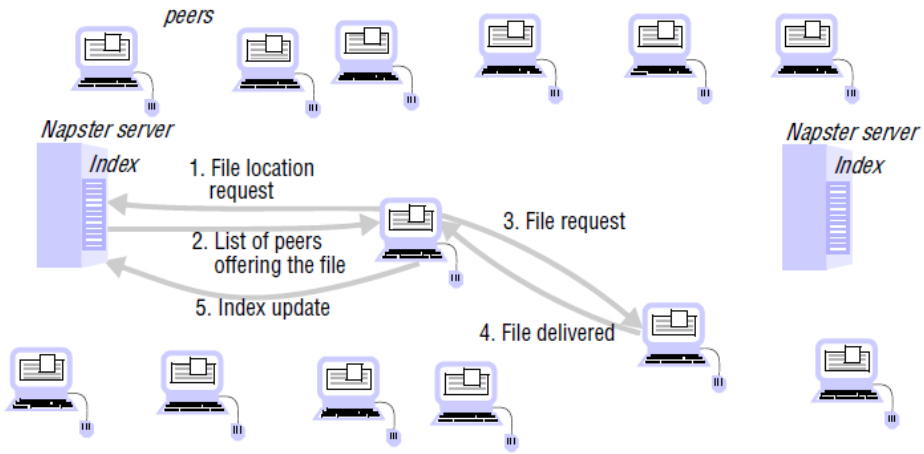

Ilustración 2 Arquitectura de índice centralizado de Napster

En su punto álgido Napster registraba picos de millones de usuarios simultáneos compartiendo y descargando ficheros de música simultáneamente. Sin embargo, como resultado de diversos procedimientos legales causado por demandas de los propietarios de copyright, Napster fue cerrado forzosamente en julio de 2001.

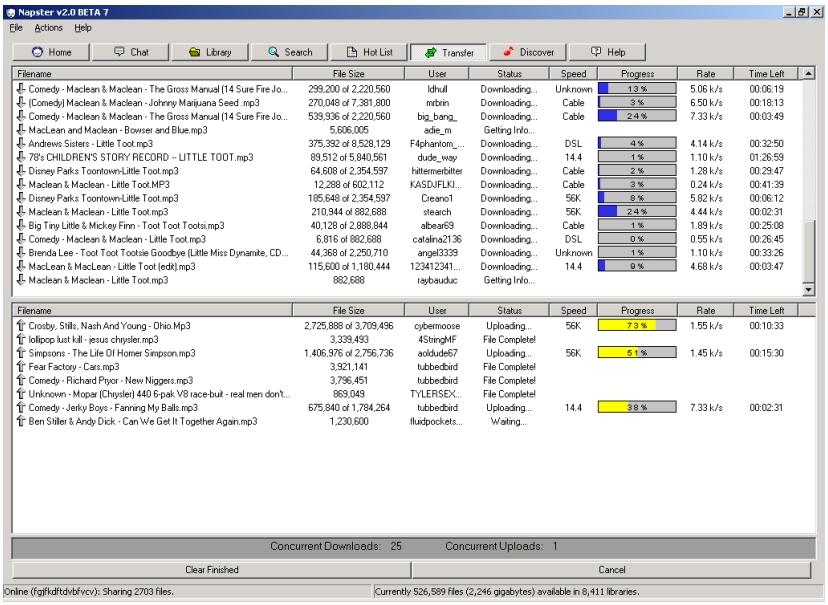

Ilustración 3 Interfaz de usuario de Napster

Esta caída del gigante P2P provocó una controversia acerca de los derechos de copyright en la compartición de archivos que se extiende hasta nuestros días.

Independientemente del carácter legal del incidente, una serie de hechos quedaron probados a raíz de la hegemonía de Napster. Se demostró que un sistema de compartición a escala global era perfectamente factible y escalable, y que los usuarios podían ser parte del mismo sistema del que requerían uso. Quedaron también patentes algunas limitaciones del sistema como la consistencia entre distintas versiones del mismo archivo alojadas en distintos dispositivos. Si bien esto no constituía un problema para Napster, ya que los ficheros de música típicamente no son alterados, sí era cierto que para otro tipo de archivos esto se volvería un problema a resolver. Además, la escalabilidad del sistema de índices de archivos de Napster podía llegar a convertirse en un cuello de botella si el sistema crecía demasiado rápido, por lo que se advirtió la necesidad de un sistema de distribución de los índices de ficheros más eficiente.

A.3 Arquitecturas

Existen tres arquitecturas principales con las que se puede conformar una red peer-to-peer:

1. Red centralizada. En este modelo toda la información circula a través de un equipo central que se encarga de conectar a los usuarios de manera transparente. El sistema de indexación de Napster era de esta índole, aunque no su sistema de distribución de ficheros. La ventaja de este sistema es que es sencillo de administrar, pero es un sistema muy limitado en escalabilidad y muy costoso de mantener, justo lo que se deseaba resolver con las redes P2P. Además son muy sensibles a errores, ya que de resultar comprometido el servidor central la red cae por completo. La inclusión de esta arquitectura como red P2P genera cierta controversia ya que la conexión entre los usuarios no resulta directa, pero es aceptada mientras sean los usuarios los que aporten los recursos del sistema (ficheros o similares).

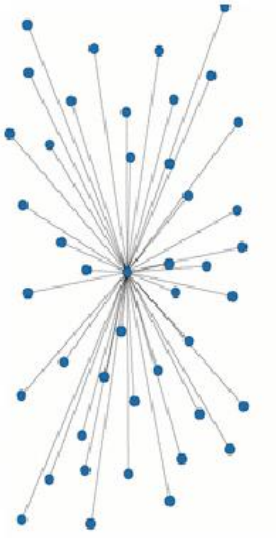

Ilustración 4 Esquema de red centralizada.

2. Red híbrida. Es un paso intermedio entre las redes centralizadas y descentralizadas. En las redes distribuidas se forman grupos de usuarios que actúan como redes centralizadas, y estas a su vez se comunican entre sí mediante un servidor central.

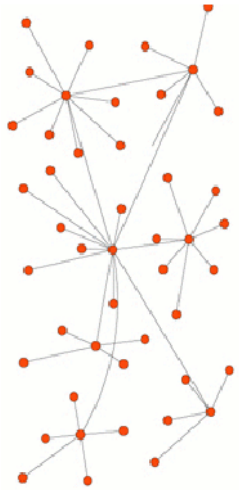

Ilustración 5 Esquema de red híbrida

Esta red escala mejor que la red centralizada pero cuenta con problemas similares a la anterior. De caer el servidor central los grupos pueden seguir comunicándose entre sí pero no con otros grupos. La administración de esta red es más trabajosa.

3. Redes descentralizadas. También llamadas redes puras, estas redes son las más comunes en la actualidad por su versatilidad y no requerir ningún tipo de gestión. En ella los nodos se comunican directamente entre sí sin la necesidad de ninguna clase de servidor. La escalabilidad de este tipo de redes es excelente.

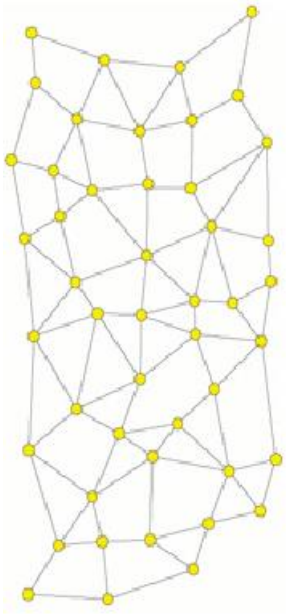

Ilustración 6 Red P2P pura

A.4 Implementaciones populares

A continuación se citan brevemente algunas aplicaciones populares de las redes P2P.

- eMule: creada en 2002 como red híbrida, se convirtió en uno de los sistemas más populares de compartición de archivos de todo tipo.

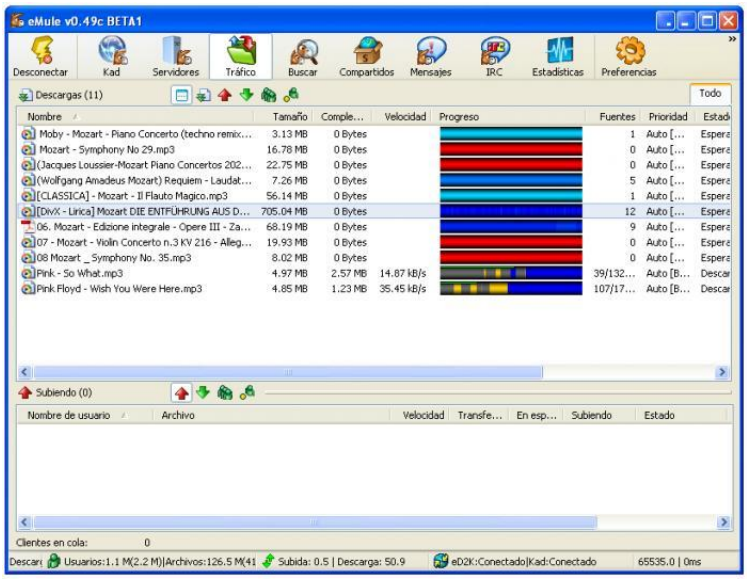

Ilustración 7 Interfaz de usuario de Emule

- BitTorrent: publicado en julio de 2001 y haciendo uso de una red P2P híbrida, BitTorrent cuenta con una tasa de más de 300 millones de usuarios mensuales de acuerdo con los datos de BitTorrent Inc. Su popularidad creció drásticamente tras el cierre del famoso sitio de descargas Megaupload.

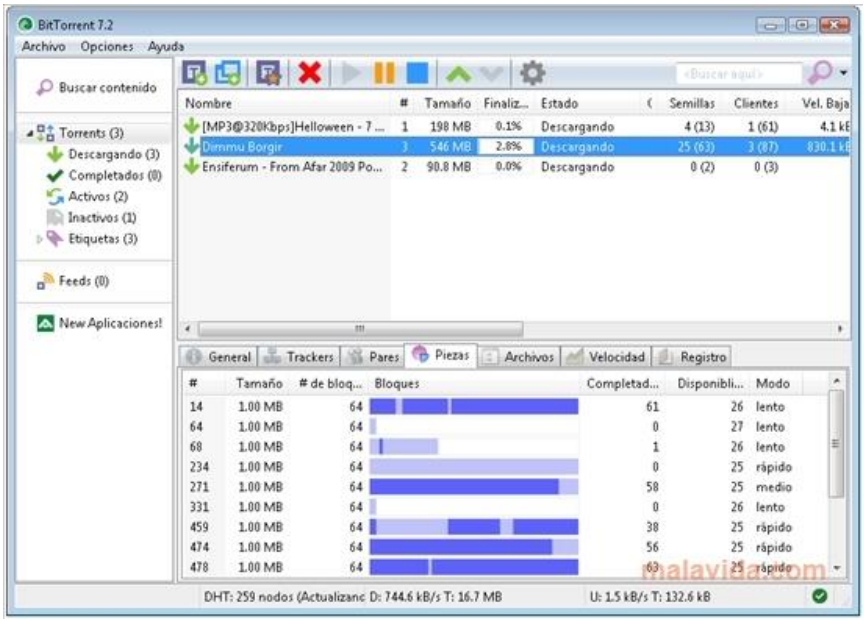

Ilustración 8 Interfaz de usuario de BitTorrent

- Ares Galaxy: creado a mediados de 2002 como red P2P descentralizada, es un software para Microsoft Windows con un diseño más elegante y atractivo para el usuario que los anteriormente citados. Además permite a los usuarios previsualizar ciertos archivos multimedia.

Ilustración 9 Interfaz de usuario de Ares

- Gnutella: haciendo uso de una red P2P pura, Gnutella surgió como alternativa descentralizada a Napster, que empleaba el sistema centralizado. La popularidad de esta aplicación aumentó significativamente tras la caída de Napster.

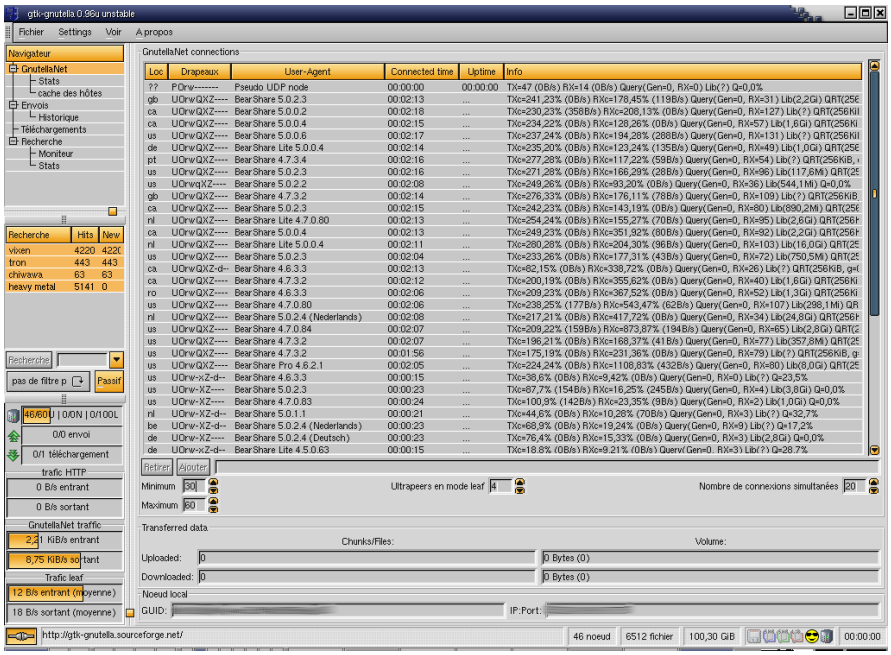

Ilustración 10 Interfaz de usuario de Gnutella

Anexo 2: INSTALACIÓN Y MODIFICACIÓN

La instalación de la aplicación solo requiere de la descarga y ejecución del fichero apk correspondiente a la misma. Es necesario contar con 18.8 MB de almacenamiento disponible en el dispositivo para poder descargar el fichero apk.

Una vez instalada la aplicación se puede proceder a su inmediato uso debido a la ausencia de cualquier clase de servidor o programa adicional con el que la aplicación necesite interactuar.

La modificación de la aplicación, por el contrario, requiere de la instalación de Android Studio y la importación del proyecto de la aplicación.

Para descargar Andoid Studio basta con acudir a la página web oficial del software[10] y seleccionar la versión del programa que se ajuste a nuestro sistema operativo:

Android Studio includes all the tools you need to build apps for Android.

DOWNLOAD ANDROID STUDIO 2.1
FOR WINDOWS (1262 MB)

VERSION: 2.1.3.0

RELEASE DATE: AUGUST 15, 2016

Select a different platform

Platform	Android Studio package	Size	SHA-1 checksum
Windows	android-studio-bundle-143.3101438-windows.exe Includes Android SDK **(recommended)**	1262 MB (1324294792 bytes)	10d319c772b80f3cb0cde952451af8429ea1b68b
	android-studio-ide-143.3101438-windows.exe No Android SDK	259 MB (271696304 bytes)	43f84de7e61f37880a126c3d567b7fa6cb90c90e
	android-studio-ide-143.3101438-windows.zip No Android SDK, no installer	275 MB (289169874 bytes)	8ad212c55c7f4dc7ab490e4b7e77ec48001ae224
Mac OS X	android-studio-ide-143.3101438-mac.dmg	273 MB (287207166 bytes)	06166759b0e1e1ee91a147dcf5227d897a184277
Linux	android-studio-ide-143.3101438-linux.zip	273 MB (286664165 bytes)	8729e6f2f1fa58f04df9f8d1caac2f5be9dfc549

Ilustración 11 Opciones de descarga de Android Studio

Una vez descargado el proceso de instalación es sencillo y guiado por el propio software.

Una vez instalado Android Studio basta con importar el proyecto incluido en el CD anexo a este documento.

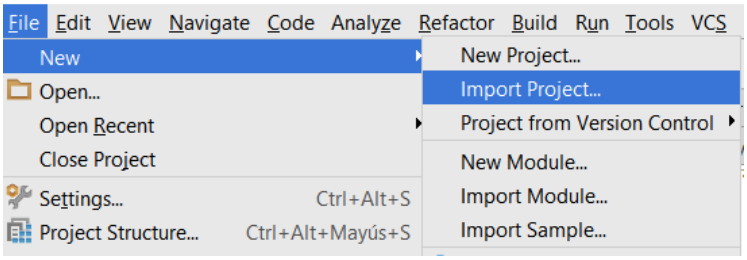

Ilustración 12 Importación del proyecto

[10] https://developer.android.com/studio/index.html

Dado que el proyecto contiene ya las librerías de AllJoyn no es necesario descargarlas ni incluirlas de nuevo en ningún lugar, pero si se desea comenzar un nuevo proyecto en el que se haga uso de AllJoyn basta con acudir a su sitio web donde tienen un apartado de descargas[11] en el que podemos seleccionar todas las funcionalidades de AllJoyn o solo algunos de sus módulos en función de lo que necesitemos para nuestra aplicación:

v15.09a

DOWNLOAD	PLATFORM	VERSION	RELEASE DATE	DOWNLOAD LINK
Thin Core Source	Source	15.09.00a	Nov 12 2015	Thin Core Source
Standard Core Source	Source	15.09.00a	Nov 12 2015	Standard Core Source
Core SDK - release	Android	15.09.00a	Nov 12 2015	Core SDK - release
Core SDK - debug	Android	15.09.00a	Nov 12 2015	Core SDK - debug

v15.09

DOWNLOAD	PLATFORM	VERSION	RELEASE DATE	DOWNLOAD LINK
Base Services Source	Source	15.09.00	Nov 17 2015	Base Services Source
Code Generator Source	Source	15.09.00	Nov 30 2015	Code Generator Source
Standard Core Source	Source	15.09.00	Sep 30 2015	Standard Core Source
Thin Core Source	Source	15.09.00	Sep 30 2015	Thin Core Source
Security Manager Source	Source	15.09.00	Sep 30 2015	Security Manager Source
Core SDK - release	Android	15.09.00	Sep 30 2015	Core SDK - release
Core SDK - debug	Android	15.09.00	Sep 30 2015	Core SDK - debug
Onboarding SDK	Android	15.09.00	Nov 17 2015	Onboarding SDK
Notification SDK	Android	15.09.00	Nov 17 2015	Notification SDK

Ilustración 13 Varias opciones de descarga de AllJoyn

Una vez descargado el paquete elegido basta con añadir dos ficheros a nuestra aplicación. El primero es 'liballjoyn_java.so', el cual debemos emplazar en un directorio de nuestro proyecto llamado 'jniLibs/armeabi':

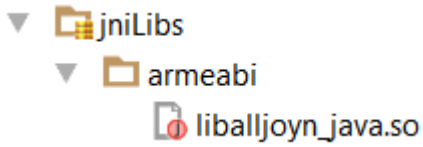

Ilustración 14 Librerías java de AllJoyn añadidas a la aplicación

El segundo es el fichero 'alljoyn.jar' con las clases del módulo que deseemos añadir. Para incluirlo en el proyecto debemos, en Android Studio, hacer click derecho sobre la carpeta de nuestro proyecto y seleccionar 'Add Support Libraries', tras lo cual seleccionaremos el fichero y quedará incluido como librería de nuestra aplicación:

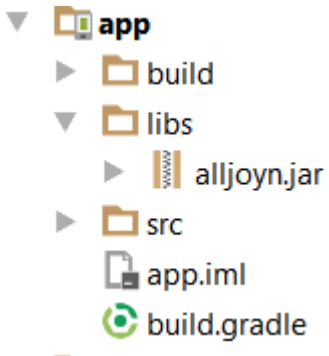

Ilustración 15 Fichero de librerías AllJoyn añadido a la aplicación

[11] https://allseenalliance.org/framework/download

Anexo 3: CÓDIGO

File - AllJoynService.java

```
1  package org.proyecto.ricardo.p2ptrain;
2
3  import android.app.Notification;
4  import android.app.PendingIntent;
5  import android.app.Service;
6  import android.content.Intent;
7  import android.os.Handler;
8  import android.os.HandlerThread;
9  import android.os.IBinder;
10 import android.os.Looper;
11 import android.os.Message;
12 import android.util.Log;
13
14 import org.alljoyn.bus.BusAttachment;
15 import org.alljoyn.bus.BusException;
16 import org.alljoyn.bus.BusListener;
17 import org.alljoyn.bus.BusObject;
18 import org.alljoyn.bus.MessageContext;
19 import org.alljoyn.bus.Mutable;
20 import org.alljoyn.bus.SessionListener;
21 import org.alljoyn.bus.SessionOpts;
22 import org.alljoyn.bus.SessionPortListener;
23 import org.alljoyn.bus.SignalEmitter;
24 import org.alljoyn.bus.Status;
25 import org.alljoyn.bus.annotation.BusSignalHandler;
26
27
28 //Clase tomada de la documentación de Alljoyn y modificada para su uso en P2PTrain.
29 //Se han mantenido los comentarios originales.
30 public class AllJoynService extends Service implements Observer {
31     private static final String TAG = "chat.AllJoynService";
32
33     /**
34      * We don't chat the bindery to communiate between any client and this
35      * service so we return null.
36      */
37     public IBinder onBind(Intent intent) {
38         Log.i(TAG, "onBind()");
39         return null;
40     }
41
42     /**
43      * Our onCreate() method is called by the Android application framework
44      * when the service is first created.  We spin up a background thread
45      * to handle any long-lived requests (pretty much all AllJoyn calls that
46      * involve communication with remote processes) that need to be done and
47      * insinuate ourselves into the custom_list of observers of the model so we can
48      * get event notifications.
49      */
50     public void onCreate() {
51         Log.i(TAG, "onCreate()");
52         startBusThread();
53         mTrainApplication = (TrainApplication)getApplication();
54         mTrainApplication.addObserver(this);
55
56         CharSequence title = "P2PTrain";
57         CharSequence message = "Red P2P conectada";
58         Intent intent = new Intent(this, TabWidget.class);
59         PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);
60         Notification notification = new Notification(R.drawable.icon, null, System.currentTimeMillis());
61         notification.setLatestEventInfo(this, title, message, pendingIntent);
62         notification.flags |= Notification.DEFAULT_SOUND | Notification.FLAG_ONGOING_EVENT | Notification.FLAG_NO_CLEAR
   ;
63
64         Log.i(TAG, "onCreate(): startForeground()");
65         startForeground(NOTIFICATION_ID, notification);
66
67         /*
68          * We have an AllJoyn handler thread running at this time, so take
69          * advantage of the fact to get connected to the bus and start finding
70          * remote channel instances in the background while the rest of the app
71          * is starting up.
72          */
73         mBackgroundHandler.connect();
74         mBackgroundHandler.startDiscovery();
75     }
76
77     private static final int NOTIFICATION_ID = 0xdefaced;
```

File - AllJoynService.java

```
78
79      /**
80       * Our onDestroy() is called by the Android appliation framework when it
81       * decides that our Service is no longer needed.  We tell our background
82       * thread to exit andremove ourselves from the custom_list of observers of the
83       * model.
84       */
85      public void onDestroy() {
86          Log.i(TAG, "onDestroy()");
87          mBackgroundHandler.cancelDiscovery();
88          mBackgroundHandler.disconnect();
89          stopBusThread();
90          mTrainApplication.deleteObserver(this);
91      }
92
93      /**
94       * The method onStartCommand() is called by the Android application
95       * framework when a client explicitly starts a Service by calling
96       * startService().  We expect that the only place this is going to be done
97       * is when the Android Application class for our application is created.
98       * The Appliation class provides our model in the sense of the MVC
99       * application we really are.
100      *
101      * We return START_STICKY to enable us to be explicity started and stopped
102      * which means that our Service will essentially run "forever" (or until
103      * Android decides that we should die for resource management issues) since
104      * our Application class is left running as long as the process is left running.
105      */
106     public int onStartCommand(Intent intent, int flags, int startId) {
107         Log.i(TAG, "onStartCommand()");
108         return START_STICKY;
109     }
110
111     /**
112      * A reference to a descendent of the Android Application class that is
113      * acting as the Model of our MVC-based application.
114      */
115     private TrainApplication mTrainApplication = null;
116
117     /**
118      * This is the event handler for the Observable/Observed design pattern.
119      * Whenever an interesting event happens in our appliation, the Model (the
120      * source of the event) notifies registered observers, resulting in this
121      * method being called since we registered as an Observer in onCreate().
122      *
123      * This method will be called in the context of the Model, which is, in
124      * turn the context of an event source.  This will either be the single
125      * Android application framework thread if the source is one of the
126      * Activities of the application or the Service.  It could also be in the
127      * context of the Service background thread.  Since the Android Application
128      * framework is a fundamentally single threaded thing, we avoid multithread
129      * issues and deadlocks by immediately getting this event into a separate
130      * execution in the context of the Service message pump.
131      *
132      * We do this by taking the event from the calling component and queueing
133      * it onto a "handler" in our Service and returning to the caller.  When
134      * the calling componenet finishes what ever caused the event notification,
135      * we expect the Android application framework to notice our pending
136      * message and run our handler in the context of the single application
137      * thread.
138      *
139      * In reality, both events are executed in the context of the single
140      * Android thread.
141      */
142     public synchronized void update(Observable o, Object arg) {
143         Log.i(TAG, "update(" + arg + ")");
144         String qualifier = (String)arg;
145
146         if (qualifier.equals(TrainApplication.APPLICATION_QUIT_EVENT)) {
147             Message message = mHandler.obtainMessage(HANDLE_APPLICATION_QUIT_EVENT);
148             mHandler.sendMessage(message);
149         }
150
151         if (qualifier.equals(TrainApplication.CHAT_JOIN_CHANNEL_EVENT)) {
152             Message message = mHandler.obtainMessage(HANDLE_USE_JOIN_CHANNEL_EVENT);
153             mHandler.sendMessage(message);
154         }
155
```

File - AllJoynService.java

```
156         if (qualifier.equals(TrainApplication.CHAT_LEAVE_CHANNEL_EVENT)) {
157             Message message = mHandler.obtainMessage(HANDLE_USE_LEAVE_CHANNEL_EVENT);
158             mHandler.sendMessage(message);
159         }
160 
161         if (qualifier.equals(TrainApplication.HOST_INIT_CHANNEL_EVENT)) {
162             Message message = mHandler.obtainMessage(HANDLE_HOST_INIT_CHANNEL_EVENT);
163             mHandler.sendMessage(message);
164         }
165 
166         if (qualifier.equals(TrainApplication.HOST_START_CHANNEL_EVENT)) {
167             Message message = mHandler.obtainMessage(HANDLE_HOST_START_CHANNEL_EVENT);
168             mHandler.sendMessage(message);
169         }
170 
171         if (qualifier.equals(TrainApplication.HOST_STOP_CHANNEL_EVENT)) {
172             Message message = mHandler.obtainMessage(HANDLE_HOST_STOP_CHANNEL_EVENT);
173             mHandler.sendMessage(message);
174         }
175 
176         if (qualifier.equals(TrainApplication.OUTBOUND_CHANGED_EVENT)) {
177 
178             Message message = mHandler.obtainMessage(HANDLE_OUTBOUND_CHANGED_EVENT);
179             mHandler.sendMessage(message);
180         }
181     }
182 
183     /**
184      * This is the Android Service message handler.  It runs in the context of the
185      * tab_main Android Service thread, which is also shared with Activities since
186      * Android is a fundamentally single-threaded system.
187      *
188      * The important thing for us is to note that this thread cannot be blocked for
189      * a significant amount of time or we risk the dreaded "force close" message.
190      * We can run relatively short-lived operations here, but we need to run our
191      * distributed system calls in a background thread.
192      *
193      * This handler serves translates from UI-related events into AllJoyn events
194      * and decides whether functions can be handled in the context of the
195      * Android tab_main thread or if they must be dispatched to a background thread
196      * which can take as much time as needed to accomplish a task.
197      */
198     private Handler mHandler = new Handler() {
199         public void handleMessage(Message msg) {
200             switch (msg.what) {
201                 case HANDLE_APPLICATION_QUIT_EVENT:
202                 {
203                     Log.i(TAG, "mHandler.handleMessage(): APPLICATION_QUIT_EVENT");
204                     mBackgroundHandler.leaveSession();
205                     mBackgroundHandler.cancelAdvertise();
206                     mBackgroundHandler.unbindSession();
207                     mBackgroundHandler.releaseName();
208                     mBackgroundHandler.exit();
209                     stopSelf();
210                 }
211                 break;
212             case HANDLE_USE_JOIN_CHANNEL_EVENT:
213                 {
214                     Log.i(TAG, "mHandler.handleMessage(): USE_JOIN_CHANNEL_EVENT");
215                     mBackgroundHandler.joinSession();
216                 }
217                 break;
218             case HANDLE_USE_LEAVE_CHANNEL_EVENT:
219                 {
220                     Log.i(TAG, "mHandler.handleMessage(): USE_LEAVE_CHANNEL_EVENT");
221                     mBackgroundHandler.leaveSession();
222                 }
223                 break;
224             case HANDLE_HOST_INIT_CHANNEL_EVENT:
225                 {
226                     Log.i(TAG, "mHandler.handleMessage(): HOST_INIT_CHANNEL_EVENT");
227                 }
228                 break;
229             case HANDLE_HOST_START_CHANNEL_EVENT:
230                 {
231                     Log.i(TAG, "mHandler.handleMessage(): HOST_START_CHANNEL_EVENT");
232                     mBackgroundHandler.requestName();
233                     mBackgroundHandler.bindSession();
```

File - AllJoynService.java

```
234                     mBackgroundHandler.advertise();
235                 }
236                 break;
237             case HANDLE_HOST_STOP_CHANNEL_EVENT:
238                 {
239                     Log.i(TAG, "mHandler.handleMessage(): HOST_STOP_CHANNEL_EVENT");
240                     mBackgroundHandler.cancelAdvertise();
241                     mBackgroundHandler.unbindSession();
242                     mBackgroundHandler.releaseName();
243                 }
244                 break;
245             case HANDLE_OUTBOUND_CHANGED_EVENT:
246                 {
247                     Log.i(TAG, "mHandler.handleMessage(): OUTBOUND_CHANGED_EVENT");
248                     mBackgroundHandler.sendMessages();
249                 }
250                 break;
251 
252             default:
253                 break;
254             }
255         }
256     };
257 
258     /**
259      * Value for the HANDLE_APPLICATION_QUIT_EVENT case observer notification handler.
260      */
261     private static final int HANDLE_APPLICATION_QUIT_EVENT = 0;
262 
263     /**
264      * Value for the HANDLE_USE_JOIN_CHANNEL_EVENT case observer notification handler.
265      */
266     private static final int HANDLE_USE_JOIN_CHANNEL_EVENT = 1;
267 
268     /**
269      * Value for the HANDLE_USE_LEAVE_CHANNEL_EVENT case observer notification handler.
270      */
271     private static final int HANDLE_USE_LEAVE_CHANNEL_EVENT = 2;
272 
273     /**
274      * Value for the HANDLE_HOST_INIT_CHANNEL_EVENT case observer notification handler.
275      */
276     private static final int HANDLE_HOST_INIT_CHANNEL_EVENT = 3;
277 
278     /**
279      * Value for the HANDLE_HOST_START_CHANNEL_EVENT case observer notification handler.
280      */
281     private static final int HANDLE_HOST_START_CHANNEL_EVENT = 4;
282 
283     /**
284      * Value for the HANDLE_HOST_STOP_CHANNEL_EVENT case observer notification handler.
285      */
286     private static final int HANDLE_HOST_STOP_CHANNEL_EVENT = 5;
287 
288     /**
289      * Value for the HANDLE_OUTBOUND_CHANGED_EVENT case observer notification handler.
290      */
291     private static final int HANDLE_OUTBOUND_CHANGED_EVENT = 6;
292 
293     /**
294      * Enumeration of the states of the AllJoyn bus attachment.  This
295      * lets us make a note to ourselves regarding where we are in the process
296      * of preparing and tearing down the fundamental connection to the AllJoyn
297      * bus.
298      *
299      * This should really be a more private think, but for the sample we want
300      * to show the user the states we are running through.  Because we are
301      * really making a data hiding exception, and because we trust ourselves,
302      * we don't go to any effort to prevent the UI from changing our state out
303      * from under us.
304      *
305      * There are separate variables describing the states of the client
306      * ("chat") and service ("host") pieces.
307      */
308     public static enum BusAttachmentState {
309         DISCONNECTED,    /** The bus attachment is not connected to the AllJoyn bus */
310         CONNECTED,        /** The  bus attachment is connected to the AllJoyn bus */
311         DISCOVERING        /** The bus attachment is discovering remote attachments hosting chat channels */
```

File - AllJoynService.java

```
312     }
313
314     /**
315      * The state of the AllJoyn bus attachment.
316      */
317     private BusAttachmentState mBusAttachmentState = BusAttachmentState.DISCONNECTED;
318
319     /**
320      * Enumeration of the states of a hosted chat channel.  This lets us make a
321      * note to ourselves regarding where we are in the process of preparing
322      * and tearing down the AllJoyn pieces responsible for providing the chat
323      * service.  In order to be out of the IDLE state, the BusAttachment state
324      * must be at least CONNECTED.
325      */
326     public static enum HostChannelState {
327         IDLE,             /** There is no hosted chat channel */
328         NAMED,             /** The well-known name for the channel has been successfully acquired */
329         BOUND,             /** A session port has been bound for the channel */
330         ADVERTISED,         /** The bus attachment has advertised itself as hosting an chat channel */
331         CONNECTED       /** At least one remote device has connected to a session on the channel */
332     }
333
334     /**
335      * The state of the AllJoyn components responsible for hosting an chat channel.
336      */
337     private HostChannelState mHostChannelState = HostChannelState.IDLE;
338
339     /**
340      * Enumeration of the states of a hosted chat channel.  This lets us make a
341      * note to ourselves regarding where we are in the process of preparing
342      * and tearing down the AllJoyn pieces responsible for providing the chat
343      * service.  In order to be out of the IDLE state, the BusAttachment state
344      * must be at least CONNECTED.
345      */
346     public static enum ChatChannelState {
347         IDLE,             /** There is no used chat channel */
348         JOINED,             /** The session for the channel has been successfully joined */
349     }
350
351     /**
352      * The state of the AllJoyn components responsible for hosting an chat channel.
353      */
354     private ChatChannelState mChatChannelState = ChatChannelState.IDLE;
355
356     /**
357      * This is the AllJoyn background thread handler class.  AllJoyn is a
358      * distributed system and must therefore make calls to other devices over
359      * networks.  These calls may take arbitrary amounts of time.  The Android
360      * application framework is fundamentally single-threaded and so the tab_main
361      * Service thread that is executing in our component is the same thread as
362      * the ones which appear to be executing the user interface code in the
363      * other Activities.  We cannot block this thread while waiting for a
364      * network to respond, so we need to run our calls in the context of a
365      * background thread.  This is the class that provides that background
366      * thread implementation.
367      *
368      * When we need to do some possibly long-lived task, we just pass a message
369      * to an object implementing this class telling it what needs to be done.
370      * There are two tab_main parts to this class:  an external API and the actual
371      * handler.  In order to make life easier for callers, we provide API
372      * methods to deal with the actual message passing, and then when the
373      * handler thread is executing the desired method, it calls out to an
374      * implementation in the enclosing class.  For example, in order to perform
375      * a connect() operation in the background, the enclosing class calls
376      * BackgroundHandler.connect(); and the result is that the enclosing class
377      * method doConnect() will be called in the context of the background
378      * thread.
379      */
380     private final class BackgroundHandler extends Handler {
381         public BackgroundHandler(Looper looper) {
382             super(looper);
383         }
384
385         /**
386          * Exit the background handler thread.  This will be the last message
387          * executed by an instance of the handler.
388          */
389         public void exit() {
```

File - AllJoynService.java

```
390             Log.i(TAG, "mBackgroundHandler.exit()");
391             Message msg = mBackgroundHandler.obtainMessage(EXIT);
392             mBackgroundHandler.sendMessage(msg);
393         }
394 
395         /**
396          * Connect the application to the Alljoyn bus attachment.  We expect
397          * this method to be called in the context of the tab_main Service thread.
398          * All this method does is to dispatch a corresponding method in the
399          * context of the service worker thread.
400          */
401         public void connect() {
402             Log.i(TAG, "mBackgroundHandler.connect()");
403             Message msg = mBackgroundHandler.obtainMessage(CONNECT);
404             mBackgroundHandler.sendMessage(msg);
405         }
406 
407         /**
408          * Disonnect the application from the Alljoyn bus attachment.  We
409          * expect this method to be called in the context of the tab_main Service
410          * thread.  All this method does is to dispatch a corresponding method
411          * in the context of the service worker thread.
412          */
413         public void disconnect() {
414             Log.i(TAG, "mBackgroundHandler.disconnect()");
415             Message msg = mBackgroundHandler.obtainMessage(DISCONNECT);
416             mBackgroundHandler.sendMessage(msg);
417         }
418 
419         /**
420          * Start discovering remote instances of the application.  We expect
421          * this method to be called in the context of the tab_main Service thread.
422          * All this method does is to dispatch a corresponding method in the
423          * context of the service worker thread.
424          */
425         public void startDiscovery() {
426             Log.i(TAG, "mBackgroundHandler.startDiscovery()");
427             Message msg = mBackgroundHandler.obtainMessage(START_DISCOVERY);
428             mBackgroundHandler.sendMessage(msg);
429         }
430 
431         /**
432          * Stop discovering remote instances of the application.  We expect
433          * this method to be called in the context of the tab_main Service thread.
434          * All this method does is to dispatch a corresponding method in the
435          * context of the service worker thread.
436          */
437         public void cancelDiscovery() {
438             Log.i(TAG, "mBackgroundHandler.stopDiscovery()");
439             Message msg = mBackgroundHandler.obtainMessage(CANCEL_DISCOVERY);
440             mBackgroundHandler.sendMessage(msg);
441         }
442 
443         public void requestName() {
444             Log.i(TAG, "mBackgroundHandler.requestName()");
445             Message msg = mBackgroundHandler.obtainMessage(REQUEST_NAME);
446             mBackgroundHandler.sendMessage(msg);
447         }
448 
449         public void releaseName() {
450             Log.i(TAG, "mBackgroundHandler.releaseName()");
451             Message msg = mBackgroundHandler.obtainMessage(RELEASE_NAME);
452             mBackgroundHandler.sendMessage(msg);
453         }
454 
455         public void bindSession() {
456             Log.i(TAG, "mBackgroundHandler.bindSession()");
457             Message msg = mBackgroundHandler.obtainMessage(BIND_SESSION);
458             mBackgroundHandler.sendMessage(msg);
459         }
460 
461         public void unbindSession() {
462             Log.i(TAG, "mBackgroundHandler.unbindSession()");
463             Message msg = mBackgroundHandler.obtainMessage(UNBIND_SESSION);
464             mBackgroundHandler.sendMessage(msg);
465         }
466 
467         public void advertise() {
```

File - AllJoynService.java

```
468             Log.i(TAG, "mBackgroundHandler.advertise()");
469             Message msg = mBackgroundHandler.obtainMessage(ADVERTISE);
470             mBackgroundHandler.sendMessage(msg);
471         }
472
473         public void cancelAdvertise() {
474             Log.i(TAG, "mBackgroundHandler.cancelAdvertise()");
475             Message msg = mBackgroundHandler.obtainMessage(CANCEL_ADVERTISE);
476             mBackgroundHandler.sendMessage(msg);
477         }
478
479         public void joinSession() {
480             Log.i(TAG, "mBackgroundHandler.joinSession()");
481             Message msg = mBackgroundHandler.obtainMessage(JOIN_SESSION);
482             mBackgroundHandler.sendMessage(msg);
483         }
484
485         public void leaveSession() {
486             Log.i(TAG, "mBackgroundHandler.leaveSession()");
487             Message msg = mBackgroundHandler.obtainMessage(LEAVE_SESSION);
488             mBackgroundHandler.sendMessage(msg);
489         }
490
491         public void sendMessages() {
492             Log.i(TAG, "mBackgroundHandler.sendMessages()");
493             Message msg = mBackgroundHandler.obtainMessage(SEND_MESSAGES);
494             mBackgroundHandler.sendMessage(msg);
495         }
496
497
498         /**
499          * The message handler for the worker thread that handles background
500          * tasks for the AllJoyn bus.
501          */
502         public void handleMessage(Message msg) {
503             switch (msg.what) {
504             case CONNECT:
505                 doConnect();
506                 break;
507             case DISCONNECT:
508                 doDisconnect();
509                 break;
510             case START_DISCOVERY:
511                 doStartDiscovery();
512                 break;
513             case CANCEL_DISCOVERY:
514                 doStopDiscovery();
515                 break;
516             case REQUEST_NAME:
517                 doRequestName();
518                 break;
519             case RELEASE_NAME:
520                 doReleaseName();
521                 break;
522             case BIND_SESSION:
523                 doBindSession();
524                 break;
525             case UNBIND_SESSION:
526                 doUnbindSession();
527                 break;
528             case ADVERTISE:
529                 doAdvertise();
530                 break;
531             case CANCEL_ADVERTISE:
532                 doCancelAdvertise();
533                 break;
534             case JOIN_SESSION:
535                 doJoinSession();
536                 break;
537             case LEAVE_SESSION:
538                 doLeaveSession();
539                 break;
540             case SEND_MESSAGES:
541                 doSendMessages();
542                 break;
543             case EXIT:
544                 getLooper().quit();
545                 break;
```

File - AllJoynService.java

```
546             default:
547                 break;
548             }
549         }
550     }
551
552     private static final int EXIT = 1;
553     private static final int CONNECT = 2;
554     private static final int DISCONNECT = 3;
555     private static final int START_DISCOVERY = 4;
556     private static final int CANCEL_DISCOVERY = 5;
557     private static final int REQUEST_NAME = 6;
558     private static final int RELEASE_NAME = 7;
559     private static final int BIND_SESSION = 8;
560     private static final int UNBIND_SESSION = 9;
561     private static final int ADVERTISE = 10;
562     private static final int CANCEL_ADVERTISE = 11;
563     private static final int JOIN_SESSION = 12;
564     private static final int LEAVE_SESSION = 13;
565     private static final int SEND_MESSAGES = 14;
566
567     /**
568      * The instance of the AllJoyn background thread handler.  It is created
569      * when Android decides the Service is needed and is called from the
570      * onCreate() method.  When Android decides our Service is no longer
571      * needed, it will call onDestroy(), which spins down the thread.
572      */
573     private BackgroundHandler mBackgroundHandler = null;
574
575     /**
576      * Since basically our whole reason for being is to spin up a thread to
577      * handle long-lived remote operations, we provide thsi method to do so.
578      */
579     private void startBusThread() {
580         HandlerThread busThread = new HandlerThread("BackgroundHandler");
581         busThread.start();
582         mBackgroundHandler = new BackgroundHandler(busThread.getLooper());
583     }
584
585     /**
586      * When Android decides that our Service is no longer needed, we need to
587      * tear down the thread that is servicing our long-lived remote operations.
588      * This method does so.
589      */
590     private void stopBusThread() {
591         mBackgroundHandler.exit();
592     }
593
594     /**
595      * The bus attachment is the object that provides AllJoyn services to Java
596      * clients.  Pretty much all communiation with AllJoyn is going to go through
597      * this obejct.
598      */
599     private BusAttachment mBus  = new BusAttachment(TrainApplication.PACKAGE_NAME, BusAttachment.RemoteMessage.Receive
    );
600
601     /**
602      * The well-known name prefix which all bus attachments hosting a channel
603      * will chat.  The NAME_PREFIX and the channel name are composed to give
604      * the well-known name a hosting bus attachment will request and
605      * advertise.
606      */
607     private static final String NAME_PREFIX = "org.proyecto.ricardo.p2ptrain";
608
609     /**
610      * The well-known session port used as the contact port for the chat service.
611      */
612     private static final short CONTACT_PORT = 27;
613
614     /**
615      * The object path used to identify the service "location" in the bus
616      * attachment.
617      */
618     private static final String OBJECT_PATH = "/p2pTrainService";
619
620     /**
621      * The TrainBusListener is a class that listens to the AllJoyn bus for
622      * notifications corresponding to the existence of events happening out on
```

File - AllJoynService.java

```
623      * the bus.  We provide one implementation of our listener to the bus
624      * attachment during the connect().
625      */
626     private class TrainBusListener extends BusListener {
627         /**
628          * This method is called when AllJoyn discovers a remote attachment
629          * that is hosting an chat channel.  We expect that since we only
630          * do a findAdvertisedName looking for instances of the chat
631          * well-known name prefix we will only find names that we know to
632          * be interesting.  When we find a remote application that is
633          * hosting a channel, we add its channel name it to the custom_list of
634          * available channels selectable by the user.
635          *
636          * In the class documentation for the BusListener note that it is a
637          * requirement for this method to be multithread safe.  This is
638          * accomplished by the chat of a monitor on the TrainApplication as
639          * exemplified by the synchronized attribute of the addFoundChannel
640          * method there.
641          */
642         public void foundAdvertisedName(String name, short transport, String namePrefix) {
643             Log.i(TAG, "mBusListener.foundAdvertisedName(" + name + ")");
644             TrainApplication application = (TrainApplication)getApplication();
645             application.addFoundChannel(name);
646         }
647 
648         /**
649          * This method is called when AllJoyn decides that a remote bus
650          * attachment that is hosting an chat channel is no longer available.
651          * When we lose a remote application that is hosting a channel, we
652          * remote its name from the custom_list of available channels selectable
653          * by the user.
654          *
655          * In the class documentation for the BusListener note that it is a
656          * requirement for this method to be multithread safe.  This is
657          * accomplished by the chat of a monitor on the TrainApplication as
658          * exemplified by the synchronized attribute of the removeFoundChannel
659          * method there.
660          */
661         public void lostAdvertisedName(String name, short transport, String namePrefix) {
662             Log.i(TAG, "mBusListener.lostAdvertisedName(" + name + ")");
663             TrainApplication application = (TrainApplication)getApplication();
664             application.removeFoundChannel(name);
665         }
666     }
667 
668     /**
669      * An instance of an AllJoyn bus listener that knows what to do with
670      * foundAdvertisedName and lostAdvertisedName notifications.  Although
671      * we often chat the anonymous class idiom when talking to AllJoyn, the
672      * bus listener works slightly differently and it is better to chat an
673      * explicitly declared class in this case.
674      */
675     private TrainBusListener mBusListener = new TrainBusListener();
676 
677     /**
678      * Implementation of the functionality related to connecting our app
679      * to the AllJoyn bus.  We expect that this method will only be called in
680      * the context of the AllJoyn bus handler thread; and while we are in the
681      * DISCONNECTED state.
682      */
683     private void doConnect() {
684         Log.i(TAG, "doConnect()");
685         org.alljoyn.bus.alljoyn.DaemonInit.PrepareDaemon(getApplicationContext());
686         assert(mBusAttachmentState == BusAttachmentState.DISCONNECTED);
687         mBus.useOSLogging(true);
688         mBus.setDebugLevel("ALLJOYN_JAVA", 7);
689         mBus.registerBusListener(mBusListener);
690 
691         /*
692          * To make a service available to other AllJoyn peers, first
693          * register a BusObject with the BusAttachment at a specific
694          * object path.  Our service is implemented by the TrainService
695          * BusObject found at the "/p2pTrainService" object path.
696          */
697         Status status = mBus.registerBusObject(mTrainService, OBJECT_PATH);
698         if (Status.OK != status) {
699             mTrainApplication.alljoynError(TrainApplication.Module.HOST, "Incapaz de registrar el bus de chat: (" +
    status + ")");
```

File - AllJoynService.java

```
700             return;
701         }
702 
703         status = mBus.connect();
704         if (status != Status.OK) {
705             mTrainApplication.alljoynError(TrainApplication.Module.GENERAL, "Incapaz de conectar al bus: (" + status +
     ")");
706             return;
707         }
708 
709         status = mBus.registerSignalHandlers(this);
710         if (status != Status.OK) {
711             mTrainApplication.alljoynError(TrainApplication.Module.GENERAL, "Incapaz de registrar manejadores de
    señales: (" + status + ")");
712             return;
713         }
714 
715         mBusAttachmentState = BusAttachmentState.CONNECTED;
716     }
717 
718     /**
719      * Implementation of the functionality related to disconnecting our app
720      * from the AllJoyn bus.  We expect that this method will only be called
721      * in the context of the AllJoyn bus handler thread.  We expect that this
722      * method will only be called in the context of the AllJoyn bus handler
723      * thread; and while we are in the CONNECTED state.
724      */
725     private boolean doDisconnect() {
726         Log.i(TAG, "doDisonnect()");
727         assert(mBusAttachmentState == BusAttachmentState.CONNECTED);
728         mBus.unregisterBusListener(mBusListener);
729         mBus.disconnect();
730         mBusAttachmentState = BusAttachmentState.DISCONNECTED;
731         return true;
732     }
733 
734     /**
735      * Implementation of the functionality related to discovering remote apps
736      * which are hosting chat channels.  We expect that this method will only
737      * be called in the context of the AllJoyn bus handler thread; and while
738      * we are in the CONNECTED state.  Since this is a core bit of functionalty
739      * for the "chat" side of the app, we always do this at startup.
740      */
741     private void doStartDiscovery() {
742         Log.i(TAG, "doStartDiscovery()");
743         assert(mBusAttachmentState == BusAttachmentState.CONNECTED);
744         Status status = mBus.findAdvertisedName(NAME_PREFIX);
745         if (status == Status.OK) {
746             mBusAttachmentState = BusAttachmentState.DISCOVERING;
747             return;
748         } else {
749             mTrainApplication.alljoynError(TrainApplication.Module.CHAT, "Incapaz de empezar a encontrar nombres
    anunciados: (" + status + ")");
750             return;
751         }
752     }
753 
754     /**
755      * Implementation of the functionality related to stopping discovery of
756      * remote apps which are hosting chat channels.
757      */
758     private void doStopDiscovery() {
759         Log.i(TAG, "doStopDiscovery()");
760         assert(mBusAttachmentState == BusAttachmentState.CONNECTED);
761         mBus.cancelFindAdvertisedName(NAME_PREFIX);
762         mBusAttachmentState = BusAttachmentState.CONNECTED;
763     }
764 
765     /**
766      * Implementation of the functionality related to requesting a well-known
767      * name from an AllJoyn bus attachment.
768      */
769     private void doRequestName() {
770         Log.i(TAG, "doRequestName()");
771 
772         /*
773          * In order to request a name, the bus attachment must at least be
774          * connected.
```

File - AllJoynService.java

```
775          */
776         int stateRelation = mBusAttachmentState.compareTo(BusAttachmentState.DISCONNECTED);
777         assert (stateRelation >= 0);
778 
779         /*
780          * We depend on the user interface and model to work together to not
781          * get this process started until a valid name is set in the channel name.
782          */
783         String wellKnownName = NAME_PREFIX + "." + mTrainApplication.hostGetChannelName();
784         Status status = mBus.requestName(wellKnownName, BusAttachment.ALLJOYN_REQUESTNAME_FLAG_DO_NOT_QUEUE);
785         if (status == Status.OK) {
786             mHostChannelState = HostChannelState.NAMED;
787             mTrainApplication.hostSetChannelState(mHostChannelState);
788         } else {
789             mTrainApplication.alljoynError(TrainApplication.Module.CHAT, "Incapaz de adquirir id: (" + status + ")");
790         }
791     }
792 
793     /**
794      * Implementation of the functionality related to releasing a well-known
795      * name from an AllJoyn bus attachment.
796      */
797     private void doReleaseName() {
798         Log.i(TAG, "doReleaseName()");
799 
800         /*
801          * In order to release a name, the bus attachment must at least be
802          * connected.
803          */
804         int stateRelation = mBusAttachmentState.compareTo(BusAttachmentState.DISCONNECTED);
805         assert (stateRelation >= 0);
806         assert(mBusAttachmentState == BusAttachmentState.CONNECTED || mBusAttachmentState == BusAttachmentState.
    DISCOVERING);
807 
808         /*
809          * We need to progress monotonically down the hosted channel states
810          * for sanity.
811          */
812         assert(mHostChannelState == HostChannelState.NAMED);
813 
814         /*
815          * We depend on the user interface and model to work together to not
816          * change the name out from under us while we are running.
817          */
818         String wellKnownName = NAME_PREFIX + "." + mTrainApplication.hostGetChannelName();
819 
820         /*
821          * There's not a lot we can do if the bus attachment refuses to release
822          * the name.  It is not a fatal error, though, if it doesn't.  This is
823          * because bus attachments can have multiple names.
824          */
825         mBus.releaseName(wellKnownName);
826         mHostChannelState = HostChannelState.IDLE;
827         mTrainApplication.hostSetChannelState(mHostChannelState);
828     }
829 
830     /**
831      * Implementation of the functionality related to binding a session port
832      * to an AllJoyn bus attachment.
833      */
834     private void doBindSession() {
835         Log.i(TAG, "doBindSession()");
836 
837         Mutable.ShortValue contactPort = new Mutable.ShortValue(CONTACT_PORT);
838         SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES, true, SessionOpts.PROXIMITY_ANY,
    SessionOpts.TRANSPORT_ANY);
839 
840         Status status = mBus.bindSessionPort(contactPort, sessionOpts, new SessionPortListener() {
841             /**
842              * This method is called when a client tries to join the session
843              * we have bound.  It asks us if we want to accept the client into
844              * our session.
845              *
846              * In the class documentation for the SessionPortListener note that
847              * it is a requirement for this method to be multithread safe.
848              * Since we never access any shared state, this requirement is met.
849              */
850             public boolean acceptSessionJoiner(short sessionPort, String joiner, SessionOpts sessionOpts) {
```

File - AllJoynService.java

```
851                 Log.i(TAG, "SessionPortListener.acceptSessionJoiner(" + sessionPort + ", " + joiner + ", " +
    sessionOpts.toString() + ")");
852
853                 /*
854                  * Accept anyone who can get our contact port correct.
855                  */
856                 if (sessionPort == CONTACT_PORT) {
857                     return true;
858                 }
859                 return false;
860             }
861
862             /**
863              * If we return true in acceptSessionJoiner, we admit a new client
864              * into our session.  The session does not really exist until a
865              * client joins, at which time the session is created and a session
866              * ID is assigned.  This method communicates to us that this event
867              * has happened, and provides the new session ID for us to chat.
868              *
869              * In the class documentation for the SessionPortListener note that
870              * it is a requirement for this method to be multithread safe.
871              * Since we never access any shared state, this requirement is met.
872              *
873              * See comments in joinSession for why the hosted chat interface is
874              * created here.
875              */
876             public void sessionJoined(short sessionPort, int id, String joiner) {
877                 Log.i(TAG, "SessionPortListener.sessionJoined(" + sessionPort + ", " + id + ", " + joiner + ")");
878                 mHostSessionId = id;
879                 SignalEmitter emitter = new SignalEmitter(mTrainService, id, SignalEmitter.GlobalBroadcast.Off);
880                 mHostTrainInterface = emitter.getInterface(TrainInterface.class);
881             }
882         });
883
884         if (status == Status.OK) {
885             mHostChannelState = HostChannelState.BOUND;
886             mTrainApplication.hostSetChannelState(mHostChannelState);
887         } else {
888             mTrainApplication.alljoynError(TrainApplication.Module.HOST, "Incapaz de conectar al puerto de sesión: ("
    + status + ")");
889             return;
890         }
891     }
892
893     /**
894      * Implementation of the functionality related to un-binding a session port
895      * from an AllJoyn bus attachment.
896      */
897     private void doUnbindSession() {
898         Log.i(TAG, "doUnbindSession()");
899
900         /*
901          * There's not a lot we can do if the bus attachment refuses to unbind
902          * our port.
903          */
904         mBus.unbindSessionPort(CONTACT_PORT);
905         mHostTrainInterface = null;
906         mHostChannelState = HostChannelState.NAMED;
907         mTrainApplication.hostSetChannelState(mHostChannelState);
908     }
909
910     /**
911      * The session identifier of the "host" session that the application
912      * provides for remote devices.  Set to -1 if not connected.
913      */
914     int mHostSessionId = -1;
915
916     /**
917      * A flag indicating that the application has joined a chat channel that
918      * it is hosting.  See the long comment in doJoinSession() for a
919      * description of this rather non-intuitively complicated case.
920      */
921     boolean mJoinedToSelf = false;
922
923     /**
924      * This is the interface over which the chat messages will be preview_squat in
925      * the case where the application is joined to a chat channel hosted
926      * by the application.  See the long comment in doJoinSession() for a
```

File - AllJoynService.java

```
927      * description of this rather non-intuitively complicated case.
928      */
929     TrainInterface mHostTrainInterface = null;
930 
931     /**
932      * Implementation of the functionality related to advertising a service on
933      * an AllJoyn bus attachment.
934      */
935     private void doAdvertise() {
936         Log.i(TAG, "doAdvertise()");
937 
938         /*
939          * We depend on the user interface and model to work together to not
940          * change the name out from under us while we are running.
941          */
942         String wellKnownName = NAME_PREFIX + "." + mTrainApplication.hostGetChannelName();
943         Status status = mBus.advertiseName(wellKnownName, SessionOpts.TRANSPORT_ANY);
944 
945         if (status == Status.OK) {
946             mHostChannelState = HostChannelState.ADVERTISED;
947             mTrainApplication.hostSetChannelState(mHostChannelState);
948         } else {
949             mTrainApplication.alljoynError(TrainApplication.Module.HOST, "Incapaz de anunciar id: (" + status + ")");
950             return;
951         }
952     }
953 
954     /**
955      * Implementation of the functionality related to canceling an advertisement
956      * on an AllJoyn bus attachment.
957      */
958     private void doCancelAdvertise() {
959         Log.i(TAG, "doCancelAdvertise()");
960 
961         /*
962          * We depend on the user interface and model to work together to not
963          * change the name out from under us while we are running.
964          */
965         String wellKnownName = NAME_PREFIX + "." + mTrainApplication.hostGetChannelName();
966         Status status = mBus.cancelAdvertiseName(wellKnownName, SessionOpts.TRANSPORT_ANY);
967 
968         if (status != Status.OK) {
969             mTrainApplication.alljoynError(TrainApplication.Module.HOST, "Incapaz de cancelar anuncio del id: (" +
    status + ")");
970             return;
971         }
972 
973         /*
974          * There's not a lot we can do if the bus attachment refuses to cancel
975          * our advertisement, so we don't bother to even get the status.
976          */
977         mHostChannelState = HostChannelState.BOUND;
978         mTrainApplication.hostSetChannelState(mHostChannelState);
979     }
980 
981 
982     /**
983      * Implementation of the functionality related to joining an existing
984      * local or remote session.
985      */
986     private void doJoinSession() {
987         Log.i(TAG, "doJoinSession()");
988 
989         /*
990          * There is a relatively non-intuitive behavior of multipoint sessions
991          * that one needs to grok in order to understand the code below.  The
992          * important thing to understand is that there can be only one endpoint
993          * for a multipoint session in a particular bus attachment.  This
994          * endpoint can be created explicitly by a call to joinSession() or
995          * implicitly by a call to bindSessionPort().  An attempt to call
996          * joinSession() on a session port we have created with bindSessionPort()
997          * will result in an error.
998          *
999          * When we call bindSessionPort(), we do an implicit joinSession() and
1000         * thus signals (which correspond to our chat messages) will begin to
1001         * flow from the hosted chat channel as soon as we begin to host a
1002         * corresponding session.
1003         *
```

File - AllJoynService.java

```
1004          * To achieve sane user interface behavior, we need to block those
1005          * signals from the implicit join done by the bind until our user joins
1006          * the bound chat channel.  If we do not do this, the chat messages
1007          * from the chat channel hosted by the application will appear in the
1008          * chat channel joined by the application.
1009          *
1010          * Since the messages flow automatically, we can accomplish this by
1011          * turning a filter on and off in the chat signal handler.  So if we
1012          * detect that we are hosting a channel, and we find that we want to
1013          * join the hosted channel we turn the filter off.
1014          *
1015          * We also need to be able to send chat messages to the hosted channel.
1016          * This means we need to point the mTrainInterface at the session ID of
1017          * the hosted session.  There is another complexity here since the
1018          * hosted session doesn't exist until a remote session has joined.
1019          * This means that we don't have a session ID to chat to create a
1020          * SignalEmitter until a remote device does a joinSession on our
1021          * hosted session.  This, in turn, means that we have to create the
1022          * SignalEmitter after we get a sessionJoined() callback in the
1023          * SessionPortListener passed into bindSessionPort().  We chose to
1024          * create the signal emitter for this case in the sessionJoined()
1025          * callback itself.  Note that this hosted channel signal emitter
1026          * must be distinct from one constructed for the usual joinSession
1027          * since a hosted channel may have a remote device do a join at any
1028          * time, even when we are joined to another session.  If they were
1029          * not separated, a remote join on the hosted session could redirect
1030          * messages from the joined session unexpectedly.
1031          *
1032          * So, to summarize, these next few lines handle a relatively complex
1033          * case.  When we host a chat channel, we do a bindSessionPort which
1034          * *enables* the creation of a session.  As soon as a remote device
1035          * joins the hosted chat channel, a session is actually created, and
1036          * the SessionPortListener sessionJoined() callback is fired.  At that
1037          * point, we create a separate SignalEmitter using the hosted session's
1038          * sessionId that we can chat to send chat messages to the channel we
1039          * are hosting.  As soon as the session comes up, we begin receiving
1040          * chat messages from the session, so we need to filter them until the
1041          * user joins the hosted chat channel.  In a separate timeline, the
1042          * user can decide to join the chat channel she is hosting.  She can
1043          * do so either before or after the corresponding session has been
1044          * created as a result of a remote device joining the hosted session.
1045          * If she joins the hosted channel before the underlying session is
1046          * created, her chat messages will be discarded.  If she does so after
1047          * the underlying session is created, there will be a session emitter
1048          * waiting to chat to send chat messages.  In either case, the signal
1049          * filter will be turned off in order to listen to remote chat
1050          * messages.
1051          */
1052         if (mHostChannelState != HostChannelState.IDLE) {
1053             if (mTrainApplication.getMyGroup()) {
1054                 mChatChannelState = ChatChannelState.JOINED;
1055                 mTrainApplication.chatSetChannelState(mChatChannelState);
1056                 mJoinedToSelf = true;
1057                 return;
1058             }
1059         }
1060         /*
1061          * We depend on the user interface and model to work together to provide
1062          * a reasonable name.
1063          */
1064         String wellKnownName = NAME_PREFIX + "." + mTrainApplication.chatGetChannelName();
1065
1066         /*
1067          * Since we can act as the host of a channel, we know what the other
1068          * side is expecting to see.
1069          */
1070         short contactPort = CONTACT_PORT;
1071         SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES, true, SessionOpts.PROXIMITY_ANY,
     SessionOpts.TRANSPORT_ANY);
1072         Mutable.IntegerValue sessionId = new Mutable.IntegerValue();
1073
1074         Status status = mBus.joinSession(wellKnownName, contactPort, sessionId, sessionOpts, new SessionListener() {
1075             /**
1076              * This method is called when the last remote participant in the
1077              * chat session leaves for some reason and we no longer have anyone
1078              * to chat with.
1079              *
1080              * In the class documentation for the BusListener note that it is a
```

File - AllJoynService.java

```
1081              * requirement for this method to be multithread safe.  This is
1082              * accomplished by the chat of a monitor on the TrainApplication as
1083              * exemplified by the synchronized attribute of the removeFoundChannel
1084              * method there.
1085              */
1086             public void sessionLost(int sessionId, int reason) {
1087                 Log.i(TAG, "BusListener.sessionLost(sessionId=" + sessionId + ",reason=" + reason + ")");
1088                 mTrainApplication.alljoynError(TrainApplication.Module.CHAT, "Se ha perdido la sesión de chat");
1089                 mChatChannelState = ChatChannelState.IDLE;
1090                 mTrainApplication.chatSetChannelState(mChatChannelState);
1091             }
1092         });
1093 
1094         if (status == Status.OK) {
1095             Log.i(TAG, "doJoinSession(): chat sessionId is " + mChatSessionId);
1096             mChatSessionId = sessionId.value;
1097         } else {
1098             mTrainApplication.alljoynError(TrainApplication.Module.CHAT, "Imposible unirse a la sesión: (" + status +
      ")");
1099             return;
1100         }
1101 
1102         SignalEmitter emitter = new SignalEmitter(mTrainService, mChatSessionId, SignalEmitter.GlobalBroadcast.Off);
1103         mTrainInterface = emitter.getInterface(TrainInterface.class);
1104 
1105         mChatChannelState = ChatChannelState.JOINED;
1106         mTrainApplication.chatSetChannelState(mChatChannelState);
1107     }
1108 
1109     /**
1110      * This is the interface over which the chat messages will be preview_squat.
1111      */
1112     TrainInterface mTrainInterface = null;
1113 
1114     /**
1115      * Implementation of the functionality related to joining an existing
1116      * remote session.
1117      */
1118     private void doLeaveSession() {
1119         Log.i(TAG, "doLeaveSession()");
1120         if (mJoinedToSelf == false) {
1121             mBus.leaveSession(mChatSessionId);
1122         }else{
1123             mTrainApplication.newLocalUserMessage("//CONTROL+KILL");
1124         }
1125         mChatSessionId = -1;
1126         mJoinedToSelf = false;
1127         mChatChannelState = ChatChannelState.IDLE;
1128         mTrainApplication.chatSetChannelState(mChatChannelState);
1129     }
1130 
1131     /**
1132      * The session identifier of the "chat" session that the application
1133      * uses to talk to remote instances.  Set to -1 if not connectecd.
1134      */
1135     int mChatSessionId = -1;
1136 
1137     /**
1138      * Implementation of the functionality related to sending messages out over
1139      * an existing remote session.  Note that we always send all of the
1140      * messages on the outbound queue, so there may be instances where this
1141      * method is called and we find nothing to send depending on the races.
1142      */
1143     private void doSendMessages() {
1144         Log.i(TAG, "doSendMessages()");
1145 
1146 
1147         String message;
1148         while ((message = mTrainApplication.getOutboundItem()) != null) {
1149             Log.i(TAG, "doSendMessages(): sending message \"" + message + "\"");
1150 
1151             /*
1152              * If we are joined to a remote session, we send the message over
1153              * the mTrainInterface.  If we are implicityly joined to a session
1154              * we are hosting, we send the message over the mHostTrainInterface.
1155              * The mHostTrainInterface may or may not exist since it is created
1156              * when the sessionJoined() callback is fired in the
1157              * SessionPortListener, so we have to check for it.
```

File - AllJoynService.java

```
1158             */
1159            try {
1160                if (mJoinedToSelf) {
1161                    if (mHostTrainInterface != null) {
1162                        mHostTrainInterface.Train(message);
1163                    }
1164                } else {
1165                    if (mTrainInterface != null) {
1166                        mTrainInterface.Train(message);
1167                    }
1168                }
1169            } catch (BusException ex) {
1170                mTrainApplication.alljoynError(TrainApplication.Module.CHAT, "Excepción de bus al enviar mensaje: ("
     + ex + ")");
1171            }
1172        }
1173    }
1174
1175
1176    /**
1177     * Our chat messages are going to be Bus Signals multicast out onto an
1178     * associated session.  In order to send signals, we need to define an
1179     * AllJoyn bus object that will allow us to instantiate a signal emmiter.
1180     */
1181    class TrainService implements TrainInterface, BusObject {
1182        /**
1183         * Intentionally empty implementation of Train method.  Since this
1184         * method is only used as a signal emitter, it will never be called
1185         * directly.
1186         */
1187        public void Train(String str) throws BusException {
1188        }
1189    }
1190
1191    /**
1192     * The TrainService is the instance of an AllJoyn interface that is exported
1193     * on the bus and allows us to send signals implementing messages
1194     */
1195    private TrainService mTrainService = new TrainService();
1196
1197    /**
1198     * The signal handler for messages received from the AllJoyn bus.
1199     *
1200     * Since the messages preview_squat on a chat channel will be preview_squat using a bus
1201     * signal, we need to provide a signal handler to receive those signals.
1202     * This is it.  Note that the name of the signal handler has the first
1203     * letter capitalized to conform with the DBus convention for signal
1204     * handler names.
1205     */
1206    @BusSignalHandler(iface = "org.proyecto.ricardo.p2ptrain", signal = "Train")
1207    public void Train(String string) {
1208
1209        /*
1210         * See the long comment in doJoinSession() for more explanation of
1211         * why this is needed.
1212         *
1213         * The only time we allow a signal from the hosted session ID to pass
1214         * through is if we are in mJoinedToSelf state.  If the source of the
1215         * signal is us, we also filter out the signal since we are going to
1216         * locally echo the signal.
1217
1218         */
1219        String uniqueName = mBus.getUniqueName();
1220        MessageContext ctx = mBus.getMessageContext();
1221        Log.i(TAG, "Chat(): chat sessionId is " + mChatSessionId);
1222        Log.i(TAG, "Chat(): message sessionId is " + ctx.sessionId);
1223
1224        /*
1225         * Always drop our own signals which may be echoed back from the system.
1226         */
1227        if (ctx.sender.equals(uniqueName)) {
1228            Log.i(TAG, "Chat(): dropped our own signal received on session " + ctx.sessionId);
1229            return;
1230        }
1231
1232        /*
1233         * Drop signals on the hosted session unless we are joined-to-self.
1234         */
```

File - AllJoynService.java

```
1235         if (mJoinedToSelf == false && ctx.sessionId == mHostSessionId) {
1236             Log.i(TAG, "Chat(): dropped signal received on hosted session " + ctx.sessionId + " when not joined-to-
     self");
1237             return;
1238         }
1239
1240         /*
1241          * We want to identify the message source somehow, so we
1242          * choose the unique name of the sender's bus attachment.
1243          */
1244         String nickname = ctx.sender;
1245         nickname = nickname.substring(nickname.length()-10, nickname.length());
1246
1247         Log.i(TAG, "Chat(): signal " + string + " received from nickname " + nickname);
1248         mTrainApplication.newRemoteUserMessage(nickname, string);
1249     }
1250
1251     /*
1252      * Load the native alljoyn_java library.  The actual AllJoyn code is
1253      * written in C++ and the alljoyn_java library provides the language
1254      * bindings from Java to C++ and vice versa.
1255      */
1256     static {
1257         Log.i(TAG, "System.loadLibrary(\"alljoyn_java\")");
1258         System.loadLibrary("alljoyn_java");
1259     }
1260 }
1261
```

File - TrainInterface.java

```java
package org.proyecto.ricardo.p2ptrain;

import org.alljoyn.bus.BusException;
import org.alljoyn.bus.annotation.BusInterface;
import org.alljoyn.bus.annotation.BusSignal;

@BusInterface (name = "org.proyecto.ricardo.p2ptrain")
public interface TrainInterface {
    /*
     * The BusSignal annotation signifies that this function should be used as
     * part of the AllJoyn interface.  The runtime is smart enough to figure
     * out that this is a used as a signal emitter and is only called to send
     * signals and not to receive signals.
     */
    @BusSignal
    public void Train(String str) throws BusException;
}

```

File - HistoricActivity.java

```java
package org.proyecto.ricardo.p2ptrain;


import android.app.Activity;
import android.app.Dialog;
import android.content.Context;
import android.os.Bundle;
import android.view.View;
import android.view.Window;
import android.view.animation.AnimationUtils;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;

import java.util.ArrayList;

//Clase que controla la pantalla de historial de ejercicios
public class HistoricActivity extends Activity {
    private DBHandler p2ptrainDB;
    private Button mDeleteButton;
    private ListView listView;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.historic);

        Context context = getApplicationContext();
        p2ptrainDB = new DBHandler(context);

        mDeleteButton = (Button) findViewById(R.id.deleteHistoric);
        mDeleteButton.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {
                showDialog(0);
            }
        });

        listView = (ListView) findViewById(R.id.historicList);
        listView.setAdapter(new HistoricListMaker(this, p2ptrainDB));

    }

    protected void onResume(){
        super.onResume();
        listView.setAdapter(new HistoricListMaker(this, p2ptrainDB));
    }

    protected Dialog onCreateDialog(int i){
        DialogBuilder builder = new DialogBuilder();
        Dialog result = builder.createHistoricDeleteDialog(HistoricActivity.this, p2ptrainDB);
        return result;
    }
}

```

File - TrainApplication.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Application;
import android.content.ComponentName;
import android.content.Intent;
import android.util.Log;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

// Clase encargada de lanzar las peticiones a AlljoynService.
// Las clases que necesitan hacer uso de los servicios de Alljoyn s suscriben
// a esta clase como Observadores. TrainApplication les mana notificaciones al
// producirse los distintos eventos que necesiten de su respuesta.
public class TrainApplication extends Application implements Observable {
    private static final String TAG = "chat.TrainApplication";
    public static String PACKAGE_NAME;

    //Cadenas utilizadas para indicar la clase de evento notificado a los observadores
    public static final String APPLICATION_QUIT_EVENT = "APPLICATION_QUIT_EVENT";
    public static final String ALLJOYN_ERROR_EVENT = "ALLJOYN_ERROR_EVENT";
    public static final String CHAT_CHANNEL_FOUND_EVENT = "CHAT_CHANNEL_FOUND_EVENT";
    public static final String HOST_CHANNEL_STATE_CHANGED_EVENT = "HOST_CHANNEL_STATE_CHANGED_EVENT";
    public static final String HOST_CHANNEL_EXERCISE_CHANGED_EVENT = "HOST_CHANNEL_EXERCISE_CHANGED_EVENT";
    public static final String CHAT_CHANNEL_STATE_CHANGED_EVENT = "CHAT_CHANNEL_STATE_CHANGED_EVENT";
    public static final String CHAT_JOIN_CHANNEL_EVENT = "CHAT_JOIN_CHANNEL_EVENT";
    public static final String CHAT_LEAVE_CHANNEL_EVENT = "CHAT_LEAVE_CHANNEL_EVENT";
    public static final String HOST_INIT_CHANNEL_EVENT = "HOST_INIT_CHANNEL_EVENT";
    public static final String HOST_START_CHANNEL_EVENT = "HOST_START_CHANNEL_EVENT";
    public static final String HOST_STOP_CHANNEL_EVENT = "HOST_STOP_CHANNEL_EVENT";
    public static final String OUTBOUND_CHANGED_EVENT = "OUTBOUND_CHANGED_EVENT";
    public static final String HISTORY_CHANGED_EVENT = "HISTORY_CHANGED_EVENT";
    public static final String CONTROL_CHANGED_EVENT = "CONTROL_CHANGED_EVENT";
    public static final String EJ_CHANGED_EVENT = "EJ_CHANGED_EVENT";

    //Número máximo de mensajes salientes apilados permitidos
    final int OUTBOUND_MAX = 5;

    //Número máximo de mensajes recibidos apilados permitidos
    final int HISTORY_MAX = 30;

    //Variable para reflejar el estado del bus. Inicialmente estamos desconectados.
    public AllJoynService.BusAttachmentState mBusAttachmentState = AllJoynService.BusAttachmentState.DISCONNECTED;

    //Variable para reflejar el estado del grupo creado por el usuario. Inicialmente está inactivo.
    private AllJoynService.HostChannelState mHostChannelState = AllJoynService.HostChannelState.IDLE;

    //Variable àra reflejar el estado del grupo al que el usuario se une.Inicialmente inactivo.
    private AllJoynService.ChatChannelState mChatChannelState = AllJoynService.ChatChannelState.IDLE;

    //Array para almacenar a los Observadores de esta clase
    private List<Observer> mObservers = new ArrayList<Observer>();

    //Array para almacenar los nombres de los grupos P2P encontrados
    private List<String> mChannels = new ArrayList<String>();

    //Array para almacenar los mensajes pendientes de ser enviados por el bus Alljoyn
    private List<String> mOutbound = new ArrayList<String>();

    //Array para almacenar los mensajes recibidos
    private List<String> mHistory = new ArrayList<String>();

    //Array que almacena los identificadores en el bus Alljoyn de otros usuarios
    private List<String> mIds = new ArrayList<String>();

    //Array que almacena los nicks de otros usuarios
    private List<String> mNicks = new ArrayList<String>();

    //Enumeración de los módulos que pueden hacer uso de TranApplication
    public static enum Module {
        NONE,
        GENERAL,
        CHAT,
        HOST
```

File - TrainApplication.java

```
79      }
80
81      //Variable para establecer la propiedad del grupo en curso
82      private boolean isMyGroup = false;
83
84      //Variable para llamar a los distintos módulos. Inicialmente, a ninguno.
85      private Module mModule = Module.NONE;
86
87      //Variables para el nombre, ejercicio y descripción y la privacidad del grupo creado por el usuario
88      private String mHostChannelName;
89      private String mHostChannelExercise;
90      private String mHostChannelDescription;
91      private boolean mHostChannelPrivate;
92
93      //Variables para el nombre, ejercicio y descripción del grupo al que el usuario se une
94      private String mChatChannelName;
95      private String mChatChannelExercise;
96      private String mChatChannelDescription;
97
98      //Cadena que almacena el último mensaje de control
99      private String mControl;
100
101     //Cadena que almacena el nombre elegido por el usuario
102     private String mNickname;
103
104     //Cadena que almacena el último error de Alljoyn
105     private String mErrorString = "ER_OK";
106
107     //Variable para llamar a AlljoynService
108     ComponentName mRunningService = null;
109
110     public void onCreate() {
111         Log.i(TAG, "onCreate()");
112         PACKAGE_NAME = getApplicationContext().getPackageName();
113         Intent intent = new Intent(this, AllJoynService.class);
114         mRunningService = startService(intent);
115         if (mRunningService == null) {
116             Log.i(TAG, "onCreate(): failed to startService()");
117         }
118     }
119
120
121     public void quit() {
122         if(chatGetChannelState() == AllJoynService.ChatChannelState.JOINED){
123             newLocalUserMessage("//CONTROL+EVENT+BYE:" +getUserNick()+" ha abandonado el grupo.");
124         }
125         notifyObservers(APPLICATION_QUIT_EVENT);
126         mRunningService = null;
127     }
128
129     //Método llamado por las distintas clases para indicar que necesitarán de los servicios de Alljoyn
130     public void checkin() {
131         Log.i(TAG, "checkin()");
132         if (mRunningService == null) {
133             Log.i(TAG, "checkin():  Starting the AllJoynService");
134             Intent intent = new Intent(this, AllJoynService.class);
135             mRunningService = startService(intent);
136             if (mRunningService == null) {
137                 Log.i(TAG, "checkin(): failed to startService()");
138             }
139         }
140     }
141
142     //Método para establecer el nombre el usuario pasado desde la clase inicial
143     public void setUserNick(String nickname){
144         mNickname = nickname;
145     }
146
147     //Método para recuperar el nombre el usuario pasado desde la clase inicial
148     public String getUserNick(){
149         return mNickname;
150     }
151
152     //Método empleado cuando Alljoyn nos devuelve un error. Se le envía a todos los observadores.
153     public synchronized void alljoynError(Module m, String s) {
154         mModule = m;
155         mErrorString = s;
156         notifyObservers(ALLJOYN_ERROR_EVENT);
```

File - TrainApplication.java

```
157     }
158
159     //Devuelve el último Módulo que haya generado error de Alljoyn
160     public Module getErrorModule() {
161         return mModule;
162     }
163
164     //Devuelve la caena con el último error de Alljoyn
165     public String getErrorString() {
166         return mErrorString;
167     }
168
169     //Método llamado cuando un nuevo grupo P2P es encontrado
170     public synchronized void addFoundChannel(String channel) {
171         Log.i(TAG, "addFoundChannel(" + channel + ")");
172         removeFoundChannel(channel);
173         mChannels.add(channel);
174         Log.i(TAG, "addFoundChannel(): added " + channel);
175         notifyObservers(CHAT_CHANNEL_FOUND_EVENT);
176     }
177
178     //Método llamado cuando un grupo P2P deja de ser anunciado
179     public synchronized void removeFoundChannel(String channel) {
180         Log.i(TAG, "removeFoundChannel(" + channel + ")");
181
182         for (Iterator<String> i = mChannels.iterator(); i.hasNext();) {
183             String string = i.next();
184             if (string.equals(channel)) {
185                 Log.i(TAG, "removeFoundChannel(): removed " + channel);
186                 i.remove();
187             }
188         }
189
190         notifyObservers(CHAT_CHANNEL_FOUND_EVENT);
191     }
192
193     //Método encargado de devolver la lista de grupos P2P encontrados
194     public synchronized List<String> getFoundChannels() {
195         Log.i(TAG, "getFoundChannels()");
196         List<String> clone = new ArrayList<String>(mChannels.size());
197         for (String string : mChannels) {
198             Log.i(TAG, "getFoundChannels(): added " + string);
199             clone.add(new String(string));
200         }
201         return clone;
202     }
203
204
205     //Método para almacenar el estado del canal creado por el usuario
206     public synchronized void hostSetChannelState(AllJoynService.HostChannelState state) {
207         mHostChannelState = state;
208         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
209     }
210
211     //Método para recuperar el estado del canal creado por el usuario
212     public synchronized AllJoynService.HostChannelState hostGetChannelState() {
213         return mHostChannelState;
214     }
215
216     //Método para establecer el nombre del grupo del usuario
217     public synchronized void hostSetChannelName(String name) {
218         mHostChannelName = name;
219         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
220     }
221
222     //Método para establecer el ejercicio del grupo del usuario
223     public synchronized void hostSetChannelExercise(String ej) {
224         mHostChannelExercise = ej;
225         if(chatGetChannelState() == AllJoynService.ChatChannelState.JOINED
226                 && getMyGroup()){
227             notifyObservers(HOST_CHANNEL_EXERCISE_CHANGED_EVENT);
228         }
229         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
230     }
231
232     //Método para establecer la descripción del grupo del usuario
233     public synchronized void hostSetChannelDescription(String name) {
234         mHostChannelDescription = name;
```

File - TrainApplication.java

```
235         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
236     }
237
238     //Método para establecer la descripción del grupo del usuario
239     public synchronized void hostSetChannelPrivate(boolean priv) {
240         mHostChannelPrivate = priv;
241     }
242
243     //Métodos para recuperar el nombre, ejercicio y descripción del grupo creado por el usuario
244     public synchronized String hostGetChannelName() {
245         return mHostChannelName;
246     }
247     public synchronized String hostGetExercise() {
248         return mHostChannelExercise;
249     }
250     public synchronized String hostGetDescription() {
251         return mHostChannelDescription;
252     }
253     public synchronized boolean hostGetPrivate() {
254         return mHostChannelPrivate;
255     }
256
257
258
259     //Método para establecer el estado del grupo al que el usuario se une
260     public synchronized void chatSetChannelState(AllJoynService.ChatChannelState state) {
261         mChatChannelState = state;
262         notifyObservers(CHAT_CHANNEL_STATE_CHANGED_EVENT);
263     }
264
265     //Método para recuperar el estado del grupo al que el usuario se ha unido
266     public synchronized AllJoynService.ChatChannelState chatGetChannelState() {
267         return mChatChannelState;
268     }
269
270     //Método para establecer el nombre del grupo al que se une el usuario
271     public synchronized void chatSetChannelName(String name) {
272         mChatChannelName = name;
273         //notifyObservers(CHAT_CHANNEL_STATE_CHANGED_EVENT);
274     }
275
276     //Método para establecer el ejercicio del grupo al que se une el usuario
277     public synchronized void chatSetChannelExercise(String name) {
278         if(name == null){
279             mChatChannelExercise = null;
280         }else{
281             mChatChannelExercise = name;
282         }
283     }
284
285     //Método para establecer la descripción del grupo al que se une el usuario
286     public synchronized void chatSetChannelDescription(String name) {
287         mChatChannelDescription = name;
288     }
289
290     //Métodos para recuperar el nombre, ejercicio y descripción del grupo creado por el usuario
291     public synchronized String chatGetChannelName() {
292         return mChatChannelName;
293     }
294     public synchronized String chatGetChannelExercise() {return mChatChannelExercise; }
295     public synchronized String chatGetChannelDescription() {return mChatChannelDescription; }
296
297
298     //Método llamado cuando el usuario indica que desea unirse a un grupo en la pestaña de grupos
299     public synchronized void chatJoinChannel() {
300         clearHistory();
301         notifyObservers(CHAT_CHANNEL_STATE_CHANGED_EVENT);
302         notifyObservers(CHAT_JOIN_CHANNEL_EVENT);
303     }
304
305     //Método llamado cuando el usuario indica que desea abandonar a un grupo
306     public synchronized void chatLeaveChannel() {
307         chatSetChannelState(AllJoynService.ChatChannelState.IDLE);
308         notifyObservers(CHAT_CHANNEL_STATE_CHANGED_EVENT);
309         notifyObservers(CHAT_LEAVE_CHANNEL_EVENT);
310     }
311
312     //Método llamado cuando se han introducido todos los valores necesarios para iniciar un grupo
```

File - TrainApplication.java

```
313     public synchronized void hostInitChannel() {
314         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
315         notifyObservers(HOST_INIT_CHANNEL_EVENT);
316     }
317
318     //Método llamado cuando el usuario inicia su propio grupo
319     public synchronized void hostStartChannel() {
320         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
321         notifyObservers(HOST_START_CHANNEL_EVENT);
322     }
323
324     //Método llamado cuando el usuario detiene su grupo
325     public synchronized void hostStopChannel() {
326         notifyObservers(HOST_CHANNEL_STATE_CHANGED_EVENT);
327         notifyObservers(HOST_STOP_CHANNEL_EVENT);
328     }
329
330     /**
331      * Método llamado al enviar un mensaje al bus Alljoyn.
332      * Inicialmente se comprueba si es un mensaje de control o de texto,
333      * tras lo cual se envía a la función addOutboundItem para que lo lance al bus.
334      * Dado que los mensajes de un usuario no son reenviados a si mismo añadimos el
335      * mensaje como un InboundItem si es de texto para mostrarlo por el chat.
336      */
337     public synchronized void newLocalUserMessage(String message) {
338         if(message.contains("//CONTROL")) {
339             mControl = message;
340         }else{
341             addInboundItem("Yo", message);
342         }
343         if (chatGetChannelState() == AllJoynService.ChatChannelState.JOINED) {
344             addOutboundItem(message);
345         }
346     }
347
348     /**
349      * Método llamado al recibir un mensaje del bus Alljoyn.
350      * Inicialmente se comprueba si es un mensaje de control, y particularmente
351      * si es un mensaje HELLO o HI.REPLY que anuncia un nuevo usuario en el grupo, del cual almacenaremos
352      * su nick. Si no es un mensaje de control se añade normalmente como un InboundItem.
353      */
354     public synchronized void newRemoteUserMessage(String nickname, String message) {
355         if(message.contains("//CONTROL")){
356             if(message.contains("EVENT")){
357
358                 if(message.contains("NEW.EX")){
359                     mControl = message;
360                     notifyObservers(CONTROL_CHANGED_EVENT);
361                 }
362
363                 String copy = message.substring(message.indexOf(":")+1,message.length());
364                 addHistoryEvent(copy);
365
366                 if(copy.contains("BYE")){
367                     mNicks.remove(mIds.indexOf(nickname));
368                     mIds.remove(mIds.indexOf(nickname));
369                 }
370
371
372             }else{
373                 if (message.contains("HI.REPLY") || message.contains("WELCOME") || message.contains("HELLO")){
374                     message = message.concat("-" + nickname);
375                 }
376                 mControl = message;
377                 notifyObservers(CONTROL_CHANGED_EVENT);
378             }
379         }else {
380             addInboundItem(nickname, message);
381         }
382     }
383
384     /**
385      * Método llamado para añadir un mensaje a la cola de salida y enviarlo.
386      * El encargado de enviarlo es AlljoynService, que recibe la notificación
387      * al estar registrado como un Observer de TrainApplication.
388      */
389     private void addOutboundItem(String message) {
390         if (mOutbound.size() == OUTBOUND_MAX) {
```

File - TrainApplication.java

```
391             mOutbound.remove(0);
392         }
393         mOutbound.add(message);
394         notifyObservers(OUTBOUND_CHANGED_EVENT);
395     }
396
397     //Método usado por AlljoynService para recuperar la lista de mensajes salientes
398     public synchronized String getOutboundItem() {
399         if (mOutbound.isEmpty()) {
400             return null;
401         } else {
402             return mOutbound.remove(0);
403         }
404     }
405
406     //Método para establecer la propiedad del grupo del usuario
407     public void setMyGroup(boolean value){
408         isMyGroup = value;
409     }
410
411     //Método para consultar la propiedad del gupo en curso
412     public boolean getMyGroup(){
413         return isMyGroup;
414     }
415
416     //Método que añade los mensajes entrantes a la cadena de mensajes recibidos
417     private void addInboundItem(String nickname, String message) {
418         addHistoryItem(nickname, message);
419     }
420
421     //Método para recuperar el último mensaje de control
422     public String getControl(){
423         return mControl;
424     }
425
426     //Método que almacena los nombres de otros usuarios junto a su identificador en el bus
427     public void updateNicks(String id, String nickname){
428         mIds.add(id);
429         mNicks.add(nickname);
430     }
431
432     //Método para limpiar la caché de usuarios
433     public void cleanNicks(){
434         mIds.clear();
435         mNicks.clear();
436     }
437
438     //Método que añade los mensajes al historial de recibidos
439     private void addHistoryItem(String nickname, String message) {
440
441         if(mIds.contains(nickname)){
442             nickname = mNicks.get(mIds.indexOf(nickname));
443         }
444
445         if (mHistory.size() == HISTORY_MAX) {
446             mHistory.remove(0);
447         }
448
449         DateFormat dateFormat = new SimpleDateFormat("HH:mm");
450         Date date = new Date();
451
452         mHistory.add("[" + dateFormat.format(date) + "] (" + nickname + ") " + message);
453
454         //Se actualiza el chat mediante una notificación al Observador ChatActivity
455         notifyObservers(HISTORY_CHANGED_EVENT);
456     }
457
458     //Método para añadir mensajes de notificación en el historial
459     public void addHistoryEvent(String message){
460         mHistory.add(message);
461         notifyObservers(HISTORY_CHANGED_EVENT);
462     }
463
464     //Método para limpiar el historial de mensajes recibidos
465     private void clearHistory() {
466         mHistory.clear();
467         notifyObservers(HISTORY_CHANGED_EVENT);
468     }
```

Page 6 of 7

File - TrainApplication.java

```
469
470     //Método para recuperar el historial de mensajes recibidos
471     public synchronized List<String> getHistory() {
472         List<String> clone = new ArrayList<String>(mHistory.size());
473         for (String string : mHistory) {
474             clone.add(new String(string));
475         }
476         return clone;
477     }
478
479     //Método utilizado por otras clases para suscribirse como Observadores
480     public synchronized void addObserver(Observer obs) {
481         Log.i(TAG, "addObserver(" + obs + ")");
482         if (mObservers.indexOf(obs) < 0) {
483             mObservers.add(obs);
484         }
485     }
486
487     //Método utilizado por otras clases para desuscribirse como Observadores
488     public synchronized void deleteObserver(Observer obs) {
489         Log.i(TAG, "deleteObserver(" + obs + ")");
490         mObservers.remove(obs);
491     }
492
493     //Método empleado para notificar a los observadores. Estos deben implementar el metodo update.
494     private void notifyObservers(Object arg) {
495         Log.i(TAG, "notifyObservers(" + arg + ")");
496         for (Observer obs : mObservers) {
497             Log.i(TAG, "notify observer = " + obs);
498             obs.update(this, arg);
499         }
500     }
501 }
502
```

File - HistoricListMaker.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseAdapter;
import android.widget.ImageView;
import android.widget.TextView;
import java.util.ArrayList;

public class HistoricListMaker extends BaseAdapter {
    private Context context;
    private final DBHandler p2ptrainDB;

    public HistoricListMaker(Context context, DBHandler db) {
        this.context = context;
        this.p2ptrainDB = db;
    }

    @Override
    public int getCount() {
        ArrayList<String[]> entries = p2ptrainDB.getEntries();
        return entries.size();
    }

    @Override
    public Object getItem(int position) {
        return null;
    }

    @Override
    public long getItemId(int position) {
        return 0;
    }


    public View getView(int position, View view, final ViewGroup parent) {
        LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);

        final ArrayList<String[]> entries = p2ptrainDB.getEntries();

        View listView;
        ImageView iview;

        if (view == null) {
            listView = inflater.inflate(R.layout.historic_list, parent, false );

            String ex = entries.get(position)[0];
            String grp = entries.get(position)[1];
            String dte = entries.get(position)[2];
            iview = (ImageView) listView.findViewById(R.id.historicImg);
            if(ex.equals("Abdominales")){
                iview.setImageResource(R.drawable.preview_abs);
            }else if(ex.equals("Flexiones")){
                iview.setImageResource(R.drawable.preview_flex);
            }else if(ex.equals("Sentadillas")){
                iview.setImageResource(R.drawable.preview_squat);
            }else if(ex.equals("Burpees")){
                iview.setImageResource(R.drawable.preview_burps);
            }else if(ex.equals("Gateadas")){
                iview.setImageResource(R.drawable.preview_cat);
            }

            TextView exercise = (TextView) listView.findViewById(R.id.historicExercise);
            exercise.setText(ex);

            TextView group = (TextView) listView.findViewById(R.id.historicGroup);
            group.setText(grp);

            TextView date = (TextView) listView.findViewById(R.id.historicDate);
            date.setText(dte);

        } else {
            listView = view;
        }
        return listView;
    }
```

File - HistoricListMaker.java

```
79 }
80
81
```

File - ExercisesListMaker.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Activity;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.TextView;

import java.lang.reflect.Array;

/*
 * Esta clase se encarga de construir los layouts de listas personalizadas,
 * tanto para el diálogo de selección de ejercicios como para la HistoricActivity.
 */
public class ExercisesListMaker extends ArrayAdapter<String>{

    private final Activity context;
    private final String[] exercises;

    private static  final String desAbs = "Ejercicio completo de abs de larga duración y múltiples variaciones.";
    private static  final String desFlex = "Ejercicio de pushups espartanas. Nivel avanzado.";
    private static  final String desSen = "Ejercicio básico y detallado de squats. Base para múltiples variaciones futuras.";
    private static  final String desBurps = "Ejercicio que combina pushups y salto. Requiere una elevada cantidad de esfuerzo.";
    private static  final String desGat = "Ejercicio de catwalk para desarrollar músculos abs, de brazos y de piernas.";

    private final String[] descriptions = {
            desAbs,
            desFlex,
            desSen,
            desBurps,
            desGat
    };
    private static final Integer[] icons ={
            R.drawable.preview_abs,
            R.drawable.preview_flex,
            R.drawable.preview_squat,
            R.drawable.preview_burps,
            R.drawable.preview_cat,
    };


    public ExercisesListMaker(Activity context, String[] exs) {
        super(context, R.layout.custom_list, exs);
        this.context = context;
        this.exercises = exs;
    }

    @Override
    public View getView(int position, View view, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View rowView= inflater.inflate(R.layout.custom_list, null, true);
        TextView txtTitle = (TextView) rowView.findViewById(R.id.title);
        TextView txtDes = (TextView) rowView.findViewById(R.id.description);
        ImageView imageView = (ImageView) rowView.findViewById(R.id.img);

        txtTitle.setText(exercises[position]);

        txtDes.setText(descriptions[position]);

        imageView.setImageResource(icons[position]);

        return rowView;
    }
}
```

File - Observer.java

```java
package org.proyecto.ricardo.p2ptrain;

//Interfaz para los objetos de tipo Observer
public interface Observer {
    public void update(Observable o, Object arg);
}

```

File - DBHandler.java

```java
package org.proyecto.ricardo.p2ptrain;


import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.DatabaseUtils;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

import java.util.ArrayList;

//Clase encargada de crear la base de datos e introducir y extraer datos de ella
public class DBHandler extends SQLiteOpenHelper {
    private static final int DB_VERSION = 1;
    private static final String DBNAME = "P2PTrainDB.db";
    private static final String TABLE ="CREATE TABLE IF NOT EXISTS Exercises " +
            " (id INTEGER PRIMARY KEY, exercise TEXT, grroup TEXT, date TEXT)";
    //El nombre de la columna está mal escrito porque 'group' es una palabra
    // reservada en SQL

    public DBHandler(Context context) {
        super(context, DBNAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS" + TABLE);
        onCreate(db);
    }

    public boolean insertEntry(int id, String exercise, String group, String date) {
        long result=0;
        SQLiteDatabase db = getWritableDatabase();
        if (db != null) {
            ContentValues values = new ContentValues();
            values.put("id", id);
            values.put("exercise", exercise);
            values.put("grroup", group);
            values.put("date", date);
            result=db.insert("Exercises", null, values);
        }
        db.close();
        return(result>0);
    }

    public ArrayList<String[]> getEntries(){
        SQLiteDatabase db = getReadableDatabase();
        ArrayList<String[]> entries = new ArrayList<String[]>();
        String[] values = {"id", "exercise", "grroup", "date"};
        Cursor m = db.query("Exercises", values, null, null, null, null, null, null);
        m.moveToFirst();
        if(m.getCount() != 0){
            do {
                String[] entry = {m.getString(1), m.getString(2), m.getString(3)};
                entries.add(entry);
            } while (m.moveToNext());
        }
        db.close();
        m.close();
        return entries;
    }

    public int getLastEntry(){
        int last;
        SQLiteDatabase db = getReadableDatabase();
        long num = DatabaseUtils.queryNumEntries(db, "Exercises");

        last = (int)num;

        return last;
    }

```

File - DBHandler.java

```
79      public void dropHistoric(){
80          SQLiteDatabase db = getWritableDatabase();
81          db.execSQL("delete from Exercises");
82      }
83  }
84
```

File - TabWidget.java

```
 1 package org.proyecto.ricardo.p2ptrain;
 2
 3 import android.app.TabActivity;
 4 import android.content.Intent;
 5 import android.content.res.Resources;
 6 import android.os.Bundle;
 7 import android.view.Window;
 8 import android.widget.TabHost;
 9
10 //Clase que da forma a las pantallas de la aplicación en forma de pestañas simultáneas
11 public class TabWidget extends TabActivity {
12     public void onCreate(Bundle savedInstanceState) {
13
14         requestWindowFeature(Window.FEATURE_NO_TITLE);
15
16         super.onCreate(savedInstanceState);
17         setContentView(R.layout.tab_main);
18
19         Resources res = getResources();
20         TabHost tabHost = getTabHost();
21         TabHost.TabSpec spec;
22         Intent intent;
23
24         //Establece la pantalla de chat en la primera pestaña
25         intent = new Intent().setClass(this, ChatActivity.class);
26         spec = tabHost.newTabSpec("chat").setIndicator("", res.getDrawable(R.drawable.ic_tab_chat)).setContent(intent);
27         tabHost.addTab(spec);
28
29         //Establece la pantalla de creación de grupos en la segunda pestaña
30         intent = new Intent().setClass(this, HostActivity.class);
31         spec = tabHost.newTabSpec("host").setIndicator("", res.getDrawable(R.drawable.ic_tab_host)).setContent(intent);
32         tabHost.addTab(spec);
33
34         //Establece la pantalla de historial en la tercera pestaña
35         intent = new Intent().setClass(this, HistoricActivity.class);
36         spec = tabHost.newTabSpec("historic").setIndicator("", res.getDrawable(R.drawable.ic_tab_record)).setContent(
   intent);
37         tabHost.addTab(spec);
38
39         //Establece la primera pestaña como la inicial
40         tabHost.setCurrentTab(0);
41     }
42 }
```

File - Observable.java

```java
package org.proyecto.ricardo.p2ptrain;

//Interfaz para los objetos tipo Observable
public interface Observable {
    public void addObserver(Observer obs);
    public void deleteObserver(Observer obs);
}

```

File - ChatActivity.java

```
1  package org.proyecto.ricardo.p2ptrain;
2
3  import android.app.Dialog;
4  import android.app.Fragment;
5  import android.app.FragmentManager;
6  import android.app.FragmentTransaction;
7  import android.content.Context;
8  import android.net.Uri;
9  import android.os.Bundle;
10 import android.os.Handler;
11 import android.os.Message;
12 import android.support.v4.app.FragmentActivity;
13 import android.util.Log;
14 import android.view.KeyEvent;
15 import android.view.View;
16 import android.view.inputmethod.EditorInfo;
17 import android.widget.ArrayAdapter;
18 import android.widget.Button;
19 import android.widget.EditText;
20 import android.widget.LinearLayout;
21 import android.widget.ListView;
22 import android.widget.TextView;
23 import android.widget.Toast;
24 import android.widget.VideoView;
25
26 import java.text.SimpleDateFormat;
27 import java.util.Date;
28 import java.util.List;
29
30 //Clase que controla la pantalla de grupos en la que se incluye el video y el chat
31 public class ChatActivity extends FragmentActivity implements Observer {
32     private static final String TAG = "chat.ChatActivity";
33
34     //Valores para el control del fragmento de vídeo
35     private final static int HIDE = 0;
36     private final static int SHOW = 1;
37
38     //Acceso a los métodos de TrainApplication
39     private TrainApplication mTrainApplication = null;
40
41     //Array para contener los mensajes de chat
42     private ArrayAdapter<String> mHistoryList;
43
44     //Cadena con el nick del usuario
45     private String nick;
46
47     //Cadena para almacenar el nombre de usuarios que quieren acceder a un grupo privado
48     private String mUserAccRequest;
49
50     //Cadena para almacenar la descripción del grupo.
51     private String mChannelDescription;
52
53     //Variables de los botones y vistas de texto
54     private Button mJoinButton;
55     private Button mLeaveButton;
56     private TextView mChannelName;
57     private TextView mChannelExercise;
58     private TextView mChannelStatus;
59     private LinearLayout mGroupInfo;
60
61     //Variables para el vídeo del ejercicio
62     private Uri uri;
63     private VideoView videoView;
64     private int secVideo;
65     private boolean isPlaying;
66     private boolean mAllSet;
67
68     //Variable de acceso a la base de datos
69     private DBHandler p2ptrainDB;
70
71     private int n;
72
73
74     public void onCreate(Bundle savedInstanceState) {
75         Log.i(TAG, "onCreate()");
76         super.onCreate(savedInstanceState);
77         setContentView(R.layout.chat);
78
```

File - ChatActivity.java

```
79          n = 0;
80
81          //Iniciamos el fragmento y lo mantenemos oculto mientras no estemos unidos a un grupo
82          FragmentManager video = getFragmentManager();
83          Fragment videoFrag = video.findFragmentById(R.id.videoFrag);
84          FragmentTransaction ft = getFragmentManager().beginTransaction();
85          ft.hide(videoFrag).commitAllowingStateLoss();
86
87          videoView = (VideoView) findViewById(R.id.vistaVideo);
88
89          //Preparamos las variables de control
90          mAllSet = false;
91          isPlaying = false;
92
93          //Preparamos el aceso a la base de datos
94          Context context = getApplicationContext();
95          p2ptrainDB = new DBHandler(context);
96
97          mHistoryList = new ArrayAdapter<String>(this, android.R.layout.test_list_item);
98          ListView hlv = (ListView) findViewById(R.id.chatHistoryList);
99          hlv.setAdapter(mHistoryList);
100
101         //Preparamos el bloque de entrada de mensajes
102         EditText messageBox = (EditText)findViewById(R.id.chatMessage);
103         messageBox.setSingleLine();
104         messageBox.setOnEditorActionListener(new TextView.OnEditorActionListener() {
105             public boolean onEditorAction(TextView view, int actionId, KeyEvent event) {
106                 if (actionId == EditorInfo.IME_ACTION_DONE) {
107                     String message = view.getText().toString();
108                     if(message.contains("//CONTROL")){
109                         message = message.replace("//CONTROL", "//C0NTR0L");
110                         //Evitamos que los usuarios generen manualmente señales de control
111                     }
112                     mTrainApplication.newLocalUserMessage(message);
113                     view.setText("");
114                 }
115                 return true;
116             }
117         });
118
119         //Preparamos los botones y textos de la actividad para su uso
120         mGroupInfo = (LinearLayout)findViewById(R.id.groupInfo);
121         mGroupInfo.setOnClickListener(new View.OnClickListener() {
122             public void onClick(View view) {
123                 Toast.makeText(getApplicationContext(), mChannelDescription,
124                             Toast.LENGTH_SHORT).show(); }
125         });
126
127         mJoinButton = (Button)findViewById(R.id.chatJoin);
128         mJoinButton.setOnClickListener(new View.OnClickListener() {
129             public void onClick(View v) {
130                 showDialog(DIALOG_JOIN_ID);
131             }
132         });
133
134         mLeaveButton = (Button)findViewById(R.id.chatLeave);
135         mLeaveButton.setOnClickListener(new View.OnClickListener() {
136             public void onClick(View v) {
137                 showDialog(DIALOG_LEAVE_ID);
138             }
139         });
140
141         mChannelName = (TextView)findViewById(R.id.chatChannelName);
142         mChannelStatus = (TextView)findViewById(R.id.chatChannelStatus);
143         mChannelExercise = (TextView)findViewById(R.id.chatChannelExercise);
144
145         //Preparamos el puntero a TrainApplication y le indicamos que estamos listos
146         mTrainApplication = (TrainApplication)getApplication();
147         mTrainApplication.checkin();
148
149         //Recogemos el nombre del usuario pasado por la actividad inicial
150         nick = mTrainApplication.getUserNick();
151
152         //Comprobamos el estado del canal
153         updateChannelState();
154         updateChannelsNumber();
155         updateHistory();
156
```

File - ChatActivity.java

```
157         //Nos registramos como Observador de TrainApplication
158         //Cuando esta reciba un evento se nos notificará
159         mTrainApplication.addObserver(this);
160     }
161
162     public void onDestroy() {
163         Log.i(TAG, "onDestroy()");
164         mTrainApplication = (TrainApplication)getApplication();
165         if(mTrainApplication.chatGetChannelState() == AllJoynService.ChatChannelState.JOINED){
166             mTrainApplication.newLocalUserMessage("//CONTROL+EVENT+BYE:" +mTrainApplication.getUserNick()+" ha
    abandonado el grupo.");
167         }
168         mTrainApplication.deleteObserver(this);
169         super.onDestroy();
170     }
171
172     /*
173      * Métodos llamados cuando el usuario pausa o reanuda el vídeo.
174      * Generan un mensaje con una señal de control enviada al bus AllJoyn.
175      */
176     public synchronized void msjPlay(){
177         secVideo = videoView.getCurrentPosition();
178         isPlaying = true;
179         mTrainApplication.newLocalUserMessage("//CONTROL+PLAY:"+Integer.toString(secVideo));
180         mTrainApplication.newLocalUserMessage("//CONTROL+EVENT: "+nick+ " ha renaudado el vídeo");
181         mTrainApplication.addHistoryEvent("Has renaudado la reproducción del vídeo");
182     }
183
184     public synchronized void msjPause(){
185         isPlaying = false;
186         mTrainApplication.newLocalUserMessage("//CONTROL+PAUSE");
187         mTrainApplication.newLocalUserMessage("//CONTROL+EVENT: "+nick+ " ha detenido el vídeo");
188         mTrainApplication.addHistoryEvent("Has detenido la reproducción del vídeo");
189     }
190
191     //Método que actualiza la lista de mensajes de chat
192     private void updateHistory() {
193         Log.i(TAG, "updateHistory()");
194         mHistoryList.clear();
195         List<String> messages = mTrainApplication.getHistory();
196         for (String message : messages) {
197             mHistoryList.add(message);
198         }
199         mHistoryList.notifyDataSetChanged();
200     }
201
202     //Método encargado de gestionar las señales de control recibidas.
203     private void manageControl(){
204         String signal = mTrainApplication.getControl();
205
206         if(signal != null){
207             if(signal.contains("PLAY")){
208                 if(!isPlaying) {
209                     String sec = signal.substring(signal.indexOf(":")+1, signal.length());
210                     videoView.seekTo(Integer.parseInt(sec));
211                     videoView.start();
212                     isPlaying = true;
213                 }
214             }else if(signal.contains("PAUSE")){
215                 if(isPlaying) {
216                     secVideo = videoView.getCurrentPosition();
217                     videoView.pause();
218                     isPlaying = false;
219                 }
220             }else if(signal.contains("HELLO")){
221                 String name = signal.substring(signal.indexOf(":")+1, signal.indexOf("-"));
222                 String id = signal.substring(signal.indexOf("-")+1, signal.length());
223                 mTrainApplication.updateNicks(id , name);
224
225                 if(mTrainApplication.getMyGroup()){
226                     if(!mTrainApplication.hostGetPrivate()){
227                         String vidstate = "paused";
228                         if(isPlaying){
229                             vidstate = "playing";
230                         }
231                         mTrainApplication.newLocalUserMessage("//CONTROL+WELCOME:"
232                                 +mTrainApplication.hostGetExercise()
233                                 +"$"+Integer.toString(secVideo)
```

File - ChatActivity.java

```
234                                 +"@"+vidstate
235                                 +"#"+mChannelDescription
236                                 +"_" +mTrainApplication.getUserNick());
237                     }else{
238                         mUserAccRequest = name;
239                         showDialog(DIALOG_ACCESS_ID);
240                     }
241                 }else{
242                     mTrainApplication.newLocalUserMessage("//CONTROL+HI.REPLY:"+mTrainApplication.getUserNick());
243                 }
244 
245                 mTrainApplication.addHistoryEvent(name+" se ha unido al grupo.");
246 
247             }else if(signal.contains("WELCOME")){
248                 if(!mAllSet){
249                     String ex = signal.substring(signal.indexOf(":")+1,signal.indexOf("$"));
250                     String sec = signal.substring(signal.indexOf("$")+1, signal.indexOf("@"));
251                     String state = signal.substring(signal.indexOf("@")+1, signal.indexOf("#"));
252                     String desc = signal.substring(signal.indexOf("#")+1, signal.indexOf("_"));
253                     String name = signal.substring(signal.indexOf("_")+1, signal.indexOf("-"));
254                     String id = signal.substring(signal.indexOf("-")+1, signal.length());
255                     mTrainApplication.chatSetChannelExercise(ex);
256                     mChannelDescription = desc;
257                     mChannelExercise.setText(ex);
258                     secVideo = Integer.parseInt(sec);
259                     if(state.equals("playing")){
260                         isPlaying = true;
261                     }else {
262                         isPlaying = false;
263                     }
264                     loadEx();
265 
266                     mTrainApplication.updateNicks(id , name);
267 
268                     mAllSet = true;
269                 }
270             }else if(signal.contains("HI.REPLY:")){
271 
272                 String name = signal.substring(signal.indexOf(":")+1,signal.indexOf("-"));
273                 String id = signal.substring(signal.indexOf("-")+1, signal.length());
274                 mTrainApplication.updateNicks(id , name);
275 
276             }else if(signal.contains("NEW.EX")){
277 
278                 isPlaying = false;
279                 int offset = "El propietario ha cambiado el ejercicio a ".length()+1;
280                 String ex = signal.substring(signal.indexOf(":")+offset,signal.length());
281                 mTrainApplication.chatSetChannelExercise(ex);
282                 updateChannelState();
283 
284             }else if(signal.contains("KILL")){
285 
286                 mTrainApplication.addHistoryEvent("El propietario ha detenido el grupo.");
287                 mTrainApplication.chatLeaveChannel();
288                 controlFrag(HIDE);
289 
290             }else if(signal.contains("KICK")){
291 
292                 String name = signal.substring(signal.indexOf(":")+1,signal.length());
293                 if(name.equals(mTrainApplication.getUserNick())){
294                     mTrainApplication.addHistoryEvent("El propietario no te ha admitido en el grupo.");
295                     mTrainApplication.chatLeaveChannel();
296                     controlFrag(HIDE);
297                 }else{
298                     mTrainApplication.addHistoryEvent("El propietario no ha admitido a "+name);
299                 }
300 
301             }
302         }
303     }
304 
305     //Método para actualizar el estado del canal mostrado al usuario
306     private void updateChannelState() {
307         Log.i(TAG, "updateHistory()");
308         AllJoynService.ChatChannelState channelState = mTrainApplication.chatGetChannelState();
309         String name = mTrainApplication.chatGetChannelName();
310         String exercise = mTrainApplication.chatGetChannelExercise();
311         mChannelDescription = mTrainApplication.chatGetChannelDescription();
```

File - ChatActivity.java

```
312
313         mChannelName.setText(name);
314         mChannelExercise.setText(exercise);
315
316         updateChannelsNumber();
317
318         switch (channelState) {
319         case IDLE:
320             mChannelStatus.setText("Ocioso");
321             mJoinButton.setEnabled(true);
322             mLeaveButton.setEnabled(false);
323
324             mChannelName.setText("");
325             mChannelExercise.setText("");
326             mChannelDescription = "";
327             mAllSet = false;
328             mTrainApplication.cleanNicks();
329             controlFrag(HIDE);
330             break;
331         case JOINED:
332             mChannelStatus.setText("Unido");
333             mJoinButton.setEnabled(false);
334             mLeaveButton.setEnabled(true);
335
336             if(!mTrainApplication.getMyGroup()){
337                 if(mAllSet){
338                     if(exercise != null){
339                         loadEx();
340                     }
341                 }else {
342                     mTrainApplication.newLocalUserMessage("//CONTROL+HELLO:"+nick);
343                 }
344             }else{
345                 mTrainApplication.chatSetChannelExercise(mTrainApplication.hostGetExercise());
346                 if(exercise != null){
347                     loadEx();
348                 }
349             }
350             controlFrag(SHOW);
351             break;
352         }
353     }
354
355     //Método encargado de actualizar el indicador numérico de grupos encontrados
356     private void updateChannelsNumber(){
357         int n = mTrainApplication.getFoundChannels().size();
358         mJoinButton.setText("Buscar grupo (" + Integer.toString(n) + ")");
359     }
360
361     //Método llamado al cambiar de ejercicio
362     private void updateExercise(){
363         mTrainApplication.chatSetChannelExercise(mTrainApplication.hostGetExercise());
364         loadEx();
365         mTrainApplication.addHistoryEvent("Has cambiado el ejercicio a "+mTrainApplication.hostGetExercise());
366         mTrainApplication.newLocalUserMessage("//CONTROL+EVENT+NEW.EX:" + "El propietario ha cambiado el ejercicio a "
    +mTrainApplication.hostGetExercise());
367     }
368
369     //Método encargado de actualizar el VideoView del ejercicio
370     private void loadEx(){
371         String exercise = mTrainApplication.chatGetChannelExercise();
372         if(exercise.compareTo("Abdominales") == 0){
373             uri = Uri.parse("android.resource://" + getPackageName() + "/"
374                     + R.raw.abs);
375         }else if(exercise.compareTo("Flexiones") == 0){
376             uri = Uri.parse("android.resource://" + getPackageName() + "/"
377                     + R.raw.pushups);
378         }else if(exercise.compareTo("Sentadillas") == 0){
379             uri = Uri.parse("android.resource://" + getPackageName() + "/"
380                     + R.raw.squats);
381         }else if(exercise.compareTo("Burpees") == 0){
382             uri = Uri.parse("android.resource://" + getPackageName() + "/"
383                     + R.raw.burpees);
384         }else if(exercise.compareTo("Gateadas") == 0){
385             uri = Uri.parse("android.resource://" + getPackageName() + "/"
386                     + R.raw.catwalk);
387         }
388
```

File - ChatActivity.java

```
389         videoView.setVideoURI(uri);
390         videoView.seekTo(secVideo);
391         if(isPlaying){
392             videoView.start();
393         }else{
394             videoView.pause();
395         }
396 
397         mChannelExercise.setText(exercise);
398 
399         int index = p2ptrainDB.getLastEntry();
400         String group = mTrainApplication.chatGetChannelName();
401         String date = new SimpleDateFormat("dd/MM/yyyy, HH:mm").format(new Date());
402         p2ptrainDB.insertEntry(index, exercise, group, date);
403     }
404 
405     //Método encargado de ocultar y mostrar el fragmento de video
406     private void controlFrag(int i){
407         FragmentManager video = getFragmentManager();
408         Fragment videoFrag = video.findFragmentById(R.id.videoFrag);
409         FragmentTransaction ft = getFragmentManager().beginTransaction();
410 
411         if (i==SHOW) {
412             ft.show(videoFrag);
413             videoView.pause();
414         } else {
415             videoView.pause();
416             isPlaying = false;
417             ft.hide(videoFrag);
418         }
419         ft.commitAllowingStateLoss();;
420     }
421 
422      /*
423       * Aquí se gestionan las llamadas a los distintos bloques de diálogo
424       * mostrados antes diversos eventos como el boton Buscar Grupos
425       */
426 
427     public static final int DIALOG_JOIN_ID = 0;
428     public static final int DIALOG_LEAVE_ID = 1;
429     public static final int DIALOG_ACCESS_ID = 2;
430     public static final int DIALOG_ALLJOYN_ERROR_ID = 3;
431 
432     protected Dialog onCreateDialog(int id) {
433         Log.i(TAG, "onCreateDialog()");
434         Dialog result = null;
435         switch(id) {
436             case DIALOG_JOIN_ID:
437             {
438                 DialogBuilder builder = new DialogBuilder();
439                 result = builder.createChatJoinDialog(this, mTrainApplication);
440             }
441             break;
442             case DIALOG_LEAVE_ID:
443             {
444                 DialogBuilder builder = new DialogBuilder();
445                 result = builder.createChatLeaveDialog(this, mTrainApplication);
446             }
447             break;
448             case DIALOG_ACCESS_ID:
449             {
450                 DialogBuilder builder = new DialogBuilder();
451                 result = builder.createChatAccessDialog(this, mTrainApplication, mUserAccRequest);
452             }
453             break;
454             case DIALOG_ALLJOYN_ERROR_ID:
455             {
456                 DialogBuilder builder = new DialogBuilder();
457                 result = builder.createAllJoynErrorDialog(this, mTrainApplication);
458             }
459             break;
460         }
461         return result;
462     }
463 
464     /*
465      * Este es el método llamado por los objetos observados (TrainApplication)
466      * cuando ocurre un evento. En función del evento indicado se responderá de
```

File - ChatActivity.java

```
467      * uno u otro modo.
468      */
469     public synchronized void update(Observable o, Object arg) {
470         Log.i(TAG, "update(" + arg + ")");
471         String qualifier = (String)arg;
472 
473         if (qualifier.equals(TrainApplication.APPLICATION_QUIT_EVENT)) {
474             Message message = mHandler.obtainMessage(HANDLE_APPLICATION_QUIT_EVENT);
475             mHandler.sendMessage(message);
476         }
477 
478         if (qualifier.equals(TrainApplication.HISTORY_CHANGED_EVENT)) {
479             Message message = mHandler.obtainMessage(HANDLE_HISTORY_CHANGED_EVENT);
480             mHandler.sendMessage(message);
481         }
482 
483         if (qualifier.equals(TrainApplication.CHAT_CHANNEL_STATE_CHANGED_EVENT)) {
484             Message message = mHandler.obtainMessage(HANDLE_CHANNEL_STATE_CHANGED_EVENT);
485             mHandler.sendMessage(message);
486         }
487 
488         if (qualifier.equals(TrainApplication.ALLJOYN_ERROR_EVENT)) {
489             Message message = mHandler.obtainMessage(HANDLE_ALLJOYN_ERROR_EVENT);
490             mHandler.sendMessage(message);
491         }
492 
493         if (qualifier.equals(TrainApplication.CONTROL_CHANGED_EVENT)) {
494             Message message = mHandler.obtainMessage(HANDLE_CONTROL_CHANGED_EVENT);
495             mHandler.sendMessage(message);
496         }
497 
498         if (qualifier.equals(TrainApplication.HOST_CHANNEL_EXERCISE_CHANGED_EVENT)) {
499             Message message = mHandler.obtainMessage(HANDLE_EXERCISE_CHANGED_EVENT);
500             mHandler.sendMessage(message);
501         }
502 
503         if (qualifier.equals(TrainApplication.CHAT_CHANNEL_FOUND_EVENT)) {
504             Message message = mHandler.obtainMessage(HANDLE_CHANNEL_FOUND_EVENT);
505             mHandler.sendMessage(message);
506         }
507     }
508 
509     /**
510      * El Handler se encarga de manejar los distintos eventos entrantes.
511      * Primero se definen las constantes que representan dichos eventos.
512     */
513 
514     private static final int HANDLE_APPLICATION_QUIT_EVENT = 0;
515     private static final int HANDLE_HISTORY_CHANGED_EVENT = 1;
516     private static final int HANDLE_CHANNEL_STATE_CHANGED_EVENT = 2;
517     private static final int HANDLE_ALLJOYN_ERROR_EVENT = 3;
518     private static final int HANDLE_CONTROL_CHANGED_EVENT = 4;
519     private static final int HANDLE_EXERCISE_CHANGED_EVENT = 5;
520     private static final int HANDLE_CHANNEL_FOUND_EVENT = 6;
521 
522 
523     private Handler mHandler = new Handler() {
524         public void handleMessage(Message msg) {
525             switch (msg.what) {
526                 case HANDLE_APPLICATION_QUIT_EVENT: {
527                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_APPLICATION_QUIT_EVENT");
528                     finish();
529                 }
530                 break;
531                 case HANDLE_HISTORY_CHANGED_EVENT: {
532                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_HISTORY_CHANGED_EVENT");
533                     updateHistory();
534                     break;
535                 }
536                 case HANDLE_CHANNEL_STATE_CHANGED_EVENT: {
537                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_CHANNEL_STATE_CHANGED_EVENT");
538                     updateChannelState();
539                     break;
540                 }
541                 case HANDLE_ALLJOYN_ERROR_EVENT: {
542                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_ALLJOYN_ERROR_EVENT");
543                     alljoynError();
544                     break;
```

File - ChatActivity.java

```
545                 }
546
547                 case HANDLE_CONTROL_CHANGED_EVENT: {
548                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_CONTROL_CHANGED_EVENT");
549                     manageControl();
550                     break;
551                 }
552
553                 case HANDLE_EXERCISE_CHANGED_EVENT: {
554                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_CONTROL_CHANGED_EVENT");
555                     updateExercise();
556                     break;
557                 }
558
559                 case HANDLE_CHANNEL_FOUND_EVENT: {
560                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_CONTROL_CHANGED_EVENT");
561                     updateChannelsNumber();
562                     break;
563                 }
564
565             default:
566                 break;
567             }
568         }
569     };
570
571     /**
572      * Método llamado cuando Alljoyn encuentra un error. Al ser esta clase la primera
573      * que se muestra se le ha asignado el manejo de errores generales.
574      */
575     private void alljoynError() {
576         if (mTrainApplication.getErrorModule() == TrainApplication.Module.GENERAL ||
577                 mTrainApplication.getErrorModule() == TrainApplication.Module.CHAT) {
578             showDialog(DIALOG_ALLJOYN_ERROR_ID);
579         }
580     }
581
582 }
```

File - HostActivity.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Activity;
import android.app.Dialog;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.TextView;

//Clase que controla la pantalla de creacion de grupos
public class HostActivity extends Activity implements Observer {
    private static final String TAG = "chat.HostActivity";

    //Acceso a los métodos de TrainApplication
    private TrainApplication mTrainApplication = null;

    //Variables de los botones y vistas de texto
    private TextView mChannelName;
    private TextView mChannelStatus;
    private TextView mChannelExercise;
    private TextView mChannelDescription;
    private Button mSetNameButton;
    private Button mExButton;
    private Button mDesButton;
    private Button mStartButton;
    private Button mStopButton;
    private Button mQuitButton;
    private CheckBox mSetPrivate;

    public void onCreate(Bundle savedInstanceState) {
        Log.i(TAG, "onCreate()");
        super.onCreate(savedInstanceState);
        setContentView(R.layout.host);

        //Preparamos los botones y textos de la actividad para su uso
        mChannelName = (TextView)findViewById(R.id.hostChannelName);
        mChannelName.setText("");

        mChannelExercise = (TextView)findViewById(R.id.hostChannelExercise);
        mChannelExercise.setText("");

        mChannelDescription = (TextView)findViewById(R.id.hostChannelDesc);
        mChannelDescription.setText("");

        mChannelStatus = (TextView)findViewById(R.id.hostChannelStatus);
        mChannelStatus.setText("Ocioso");

        mSetNameButton = (Button)findViewById(R.id.hostSetName);
        mSetNameButton.setEnabled(true);
        mSetNameButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {showDialog(DIALOG_SET_NAME_ID);
            }
        });

        mExButton = (Button)findViewById(R.id.hostSetExercise);
        mExButton.setEnabled(true);
        mExButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialog(DIALOG_SET_EX);
            }
        });

        mDesButton = (Button) findViewById(R.id.hostSetDescription);
        mDesButton.setEnabled(true);
        mDesButton.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                showDialog(DIALOG_SET_DESC);
            }
        });

        mSetPrivate = (CheckBox) findViewById(R.id.hostSetPrivate);
        mSetPrivate.setEnabled(true);

        mStartButton = (Button)findViewById(R.id.hostStart);
```

File - HostActivity.java

```
79          mStartButton.setEnabled(false);
80          mStartButton.setOnClickListener(new View.OnClickListener() {
81              public void onClick(View v) {
82                  showDialog(DIALOG_START_ID);
83              }
84          });
85
86          mStopButton = (Button)findViewById(R.id.hostStop);
87          mStopButton.setEnabled(false);
88          mStopButton.setOnClickListener(new View.OnClickListener() {
89              public void onClick(View v) {
90                  showDialog(DIALOG_STOP_ID);
91              }
92          });
93
94          mQuitButton = (Button)findViewById(R.id.hostQuit);
95          mQuitButton.setEnabled(true);
96          mQuitButton.setOnClickListener(new View.OnClickListener() {
97              public void onClick(View v) {
98                  mTrainApplication.quit();
99              }
100         });
101
102         //Preparamos el puntero a TrainApplication y le indicamos que estamos listos
103         mTrainApplication = (TrainApplication)getApplication();
104         mTrainApplication.checkin();
105
106        //Actualizamos el estado del canal
107         updateChannelState();
108
109         //Nos registramos como Observador de TrainApplication
110         //Cuando esta reciba un evento se nos notificará
111         mTrainApplication.addObserver(this);
112     }
113
114     public void onDestroy() {
115         Log.i(TAG, "onDestroy()");
116         mTrainApplication = (TrainApplication)getApplication();
117         mTrainApplication.deleteObserver(this);
118         super.onDestroy();
119     }
120
121     //Método para comprobar si el grupo a crear es privado o no
122     public boolean checkPrivate(){
123         if(mSetPrivate.isChecked()){
124             return true;
125         }else{
126             return false;
127         }
128     }
129
130     /*
131     * Este es el método llamado por los objetos observados (TrainApplication)
132     * cuando ocurre un evento. En función del evento indicado se responderá de
133     * uno u otro modo.
134     */
135     public synchronized void update(Observable o, Object arg) {
136         Log.i(TAG, "update(" + arg + ")");
137         String qualifier = (String)arg;
138
139         if (qualifier.equals(TrainApplication.APPLICATION_QUIT_EVENT)) {
140             Message message = mHandler.obtainMessage(HANDLE_APPLICATION_QUIT_EVENT);
141             mHandler.sendMessage(message);
142         }
143
144         if (qualifier.equals(TrainApplication.HOST_CHANNEL_STATE_CHANGED_EVENT)) {
145             Message message = mHandler.obtainMessage(HANDLE_CHANNEL_STATE_CHANGED_EVENT);
146             mHandler.sendMessage(message);
147         }
148
149         if (qualifier.equals(TrainApplication.ALLJOYN_ERROR_EVENT)) {
150             Message message = mHandler.obtainMessage(HANDLE_ALLJOYN_ERROR_EVENT);
151             mHandler.sendMessage(message);
152         }
153     }
154
155     //Método para actualizar el estado del canal mostrado al usuario en el bloque superior
156     private void updateChannelState() {
```

File - HostActivity.java

```
157
158         AllJoynService.HostChannelState channelState = mTrainApplication.hostGetChannelState();
159         String name = mTrainApplication.hostGetChannelName();
160         String exercise = mTrainApplication.hostGetExercise();
161         String description = mTrainApplication.hostGetDescription();
162
163         boolean haveName = true;
164         if (name == null) {
165             haveName = false;
166             name = "";
167         }
168
169         boolean haveEx = true;
170         if (exercise == null) {
171             haveEx = false;
172             exercise = "";
173         }
174
175         if (description == null) {
176             description = "";
177         }
178
179         mChannelName.setText(name);
180         mChannelExercise.setText(exercise);
181         mChannelDescription.setText(description);
182
183         switch (channelState) {
184         case IDLE:
185             mChannelStatus.setText("Ocioso");
186             break;
187         case NAMED:
188             mChannelStatus.setText("Nombre asignado");
189             break;
190         case BOUND:
191             mChannelStatus.setText("Anclado");
192             break;
193         case ADVERTISED:
194             mChannelStatus.setText("Anunciado");
195             break;
196         case CONNECTED:
197             mChannelStatus.setText("Conectado");
198             break;
199         default:
200             mChannelStatus.setText("Desconocido");
201             break;
202         }
203
204         if (channelState == AllJoynService.HostChannelState.IDLE) {
205             mSetNameButton.setEnabled(true);
206             mDesButton.setEnabled(true);
207             if (haveName && haveEx) {
208                 mStartButton.setEnabled(true);
209             } else {
210                 mStartButton.setEnabled(false);
211             }
212             mStopButton.setEnabled(false);
213         } else {
214             mSetNameButton.setEnabled(false);
215             mDesButton.setEnabled(false);
216             mSetPrivate.setEnabled(false);
217             mStartButton.setEnabled(false);
218             mStopButton.setEnabled(true);
219         }
220     }
221
222     /*
223      * Aquí se gestionan las llamadas a los distintos bloques de diálogo
224      * mostrados antes diversos eventos como el boton Buscar Grupos
225     */
226
227     static final int DIALOG_SET_NAME_ID = 0;
228     static final int DIALOG_START_ID = 1;
229     static final int DIALOG_STOP_ID = 2;
230     public static final int DIALOG_ALLJOYN_ERROR_ID = 3;
231     static final int DIALOG_SET_EX = 4;
232     static final int DIALOG_SET_DESC = 5;
233
234     protected Dialog onCreateDialog(int id) {
```

File - HostActivity.java

```
235         Log.i(TAG, "onCreateDialog()");
236         Dialog result = null;
237         switch(id) {
238             case DIALOG_SET_NAME_ID:
239             {
240                 DialogBuilder builder = new DialogBuilder();
241                 result = builder.createHostNameDialog(this, mTrainApplication);
242             }
243             break;
244             case DIALOG_START_ID:
245             {
246                 DialogBuilder builder = new DialogBuilder();
247                 result = builder.createHostStartDialog(this, mTrainApplication);
248             }
249             break;
250             case DIALOG_STOP_ID:
251             {
252                 DialogBuilder builder = new DialogBuilder();
253                 result = builder.createHostStopDialog(this, mTrainApplication);
254             }
255             break;
256             case DIALOG_ALLJOYN_ERROR_ID:
257             {
258                 DialogBuilder builder = new DialogBuilder();
259                 result = builder.createAllJoynErrorDialog(this, mTrainApplication);
260             }
261             break;
262             case DIALOG_SET_EX:
263             {
264                 DialogBuilder builder = new DialogBuilder();
265                 result = builder.createHostExerciseDialog(this, mTrainApplication);
266             }
267             break;
268             case DIALOG_SET_DESC:
269             {
270                 DialogBuilder builder = new DialogBuilder();
271                 result = builder.createHostDescriptionDialog(this, mTrainApplication);
272             }
273             break;
274         }
275 
276         return result;
277     }
278 
279     /**
280      * El Handler se encarga de manejar los distintos eventos entrantes.
281      * Primero se definen las constantes que representan dichos eventos.
282      */
283 
284     private static final int HANDLE_APPLICATION_QUIT_EVENT = 0;
285     private static final int HANDLE_CHANNEL_STATE_CHANGED_EVENT = 1;
286     private static final int HANDLE_ALLJOYN_ERROR_EVENT = 2;
287 
288     private Handler mHandler = new Handler() {
289         public void handleMessage(Message msg) {
290             switch (msg.what) {
291             case HANDLE_APPLICATION_QUIT_EVENT:
292                 {
293                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_APPLICATION_QUIT_EVENT");
294                     finish();
295                 }
296                 break;
297             case HANDLE_CHANNEL_STATE_CHANGED_EVENT:
298                 {
299                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_CHANNEL_STATE_CHANGED_EVENT");
300                     updateChannelState();
301                 }
302                 break;
303             case HANDLE_ALLJOYN_ERROR_EVENT:
304                 {
305                     Log.i(TAG, "mHandler.handleMessage(): HANDLE_ALLJOYN_ERROR_EVENT");
306                     alljoynError();
307                 }
308                 break;
309             default:
310                 break;
311             }
312         }
```

File - HostActivity.java

```
313     };
314
315     // Método llamado cuando Alljoyn encuentra un error.
316     private void alljoynError() {
317         if (mTrainApplication.getErrorModule() == TrainApplication.Module.GENERAL ||
318                 mTrainApplication.getErrorModule() == TrainApplication.Module.CHAT) {
319             showDialog(DIALOG_ALLJOYN_ERROR_ID);
320         }
321     }
322
323 }
```

File - MainActivity.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.view.View;
import android.view.Window;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.view.animation.RotateAnimation;
import android.view.inputmethod.InputMethodManager;
import android.widget.*;

import java.util.Timer;
import java.util.TimerTask;

//Clase encargada de controlar la pantalla de inicio de la aplicación
public class MainActivity extends Activity {
    //Valores para la animación de giro del icono central
    private static final float ROTATE_FROM = 0.0f;
    private static final float ROTATE_TO = -10.0f * 360.0f;

    public String nick;
    private Animation shake;
    private TextView tv;
    private EditText nickname;

    private TrainApplication mTrainApplication;

    public void onCreate(Bundle savedInstanceState) {

        requestWindowFeature(Window.FEATURE_NO_TITLE);

        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        //Animación de texto agitado al introducir un nick inválido
        shake = AnimationUtils.loadAnimation(this, R.anim.animationshake);
        tv = (TextView) findViewById(R.id.alertNick);
        nickname = (EditText) findViewById(R.id.Nick);
    }

    //Método llamado al clickar el icono central para seguir a la próxima pantalla
    public void clickIcon(View view){

        nick = nickname.getText().toString();
        if(!nick.isEmpty() && nick.length() >= 3 && nick.length() <= 12) {
            ImageView favicon = (ImageView) findViewById(R.id.mainIcon);

            RotateAnimation r;
            r = new RotateAnimation(ROTATE_FROM, ROTATE_TO, Animation.RELATIVE_TO_SELF, 0.5f, Animation.
RELATIVE_TO_SELF, 0.5f);
            r.setDuration((long) 2 * 1500);
            r.setRepeatCount(0);
            favicon.startAnimation(r);

            MediaPlayer player = MediaPlayer.create(this, R.raw.introsound);
            player.start();

            tv.setVisibility(View.GONE);
            nickname.setFocusable(false);

            InputMethodManager imm = (InputMethodManager)getSystemService(Context.INPUT_METHOD_SERVICE);
            imm.hideSoftInputFromWindow(view.getWindowToken(), 0);

            mTrainApplication = (TrainApplication)getApplication();
            mTrainApplication.setUserNick(nick);

            new Timer().schedule(new TimerTask() {
                public void run() {
                    startActivity(new Intent(MainActivity.this, TabWidget.class));
                    finish();
                }
            }, 3000);
        }else{
            tv.setVisibility(View.VISIBLE);
```

File - MainActivity.java

```
78              tv.startAnimation(shake);
79          }
80      }
81  }
82
```

File - DialogBuilder.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Activity;
import android.app.Dialog;
import android.os.SystemClock;
import android.util.Log;
import android.view.KeyEvent;
import android.view.View;
import android.view.inputmethod.EditorInfo;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.TextView;

import java.util.List;


//Clase encargada de construitr os distintos diálogos emergentes de la aplicación
public class DialogBuilder {

    private static final String TAG = "p2ptrain.Dialogs";

    private String[] listaEjercicios ={
            "Abdominales", "Flexiones", "Sentadillas", "Burpees", "Gateadas"};


    //Ese método construye el dialogo emergente al pulsar el boton Buscar Grupo en ChatActivity
    public Dialog createChatJoinDialog(final ChatActivity activity, final TrainApplication application) {
        final Dialog dialog = new Dialog(activity);
        dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
        dialog.setContentView(R.layout.chat_join_dialog);

        application.setMyGroup(false);

        ArrayAdapter<String> channelListAdapter = new ArrayAdapter<String>(activity, android.R.layout.test_list_item);
        final ListView channelList = (ListView)dialog.findViewById(R.id.chatJoinChannelList);
        channelList.setAdapter(channelListAdapter);

        List<String> channels = application.getFoundChannels();
        for (String channel : channels) {
            int lastDot = channel.lastIndexOf('.');
            if (lastDot < 0) {
                continue;
            }
            channelListAdapter.add(channel.substring(lastDot + 1));
        }
        channelListAdapter.notifyDataSetChanged();

        channelList.setOnItemClickListener(new ListView.OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
                String name = channelList.getItemAtPosition(position).toString();
                application.chatSetChannelName(name);
                application.chatJoinChannel();
                activity.removeDialog(ChatActivity.DIALOG_JOIN_ID);
            }
        });

        Button cancel = (Button)dialog.findViewById(R.id.chatJoinCancel);
        cancel.setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {
                activity.removeDialog(ChatActivity.DIALOG_JOIN_ID);
            }
        });

        return dialog;
    }

    //Este método construye el diálogo emergente al pulasr el botón Dejar Grupo de ChatActivity
    public Dialog createChatLeaveDialog(ChatActivity activity, final TrainApplication application) {
        final Dialog dialog = new Dialog(activity);
        dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
        dialog.setContentView(R.layout.chat_leave_dialog);

        Button yes = (Button)dialog.findViewById(R.id.chatLeaveOk);
        yes.setOnClickListener(new View.OnClickListener() {
            public void onClick(View view) {
```

File - DialogBuilder.java

```
79
80                  application.newLocalUserMessage("//CONTROL+EVENT:"+application.getUserNick()+" ha abandonado el grupo
    .");
81
82                  if(application.getMyGroup()){
83                      application.newLocalUserMessage("//CONTROL+KILL");
84                      application.hostStopChannel();
85                  }
86                  application.chatSetChannelExercise(null);
87                  application.chatSetChannelDescription("");
88                  application.chatLeaveChannel();
89                  dialog.cancel();
90              }
91          });
92
93          Button no = (Button)dialog.findViewById(R.id.chatLeaveCancel);
94          no.setOnClickListener(new View.OnClickListener() {
95              public void onClick(View view) {
96                  dialog.cancel();
97              }
98          });
99
100         return dialog;
101     }
102
103     public Dialog createChatAccessDialog(final ChatActivity activity, final TrainApplication application,
104                                          final String user) {
105         final Dialog dialog = new Dialog(activity);
106         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
107         dialog.setContentView(R.layout.chat_access_dialog);
108
109         TextView text = (TextView) dialog.findViewById(R.id.chatUserAccess);
110         text.setText(user+" desea unirse al grupo.");
111
112         Button yes = (Button)dialog.findViewById(R.id.chatAccessOk);
113         yes.setOnClickListener(new View.OnClickListener() {
114             public void onClick(View view) {
115                 dialog.cancel();
116             }
117         });
118
119         Button no = (Button)dialog.findViewById(R.id.chatAccessCancel);
120         no.setOnClickListener(new View.OnClickListener(){
121             public void onClick(View view) {
122                 application.newLocalUserMessage("//CONTROL+KICK:"+user);
123                 dialog.cancel();
124             }
125         });
126
127         return dialog;
128     }
129
130     //Este método construye el diálogo emergente necesario para introducir el nombre del grupo en HosActivity
131     public Dialog createHostNameDialog(HostActivity activity, final TrainApplication application) {
132         Log.i(TAG, "createHostNameDialog()");
133         final Dialog dialog = new Dialog(activity);
134         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
135         dialog.setContentView(R.layout.host_name_dialog);
136
137         final EditText channel = (EditText)dialog.findViewById(R.id.hostNameChannel);
138         channel.setOnEditorActionListener(new TextView.OnEditorActionListener() {
139             public boolean onEditorAction(TextView view, int actionId, KeyEvent event) {
140                 if (actionId == EditorInfo.IME_NULL && event.getAction() == KeyEvent.ACTION_UP) {
141                     String name = view.getText().toString();
142                     application.hostSetChannelName(name);
143                     application.hostInitChannel();
144                     dialog.cancel();
145                 }
146                 return true;
147             }
148         });
149
150         Button okay = (Button)dialog.findViewById(R.id.hostNameOk);
151         okay.setOnClickListener(new View.OnClickListener() {
152             public void onClick(View view) {
153                 String name = channel.getText().toString();
154                 application.hostSetChannelName(name);
155                 application.hostInitChannel();
```

File - DialogBuilder.java

```
156                 dialog.cancel();
157             }
158         });
159 
160         Button cancel = (Button)dialog.findViewById(R.id.hostNameCancel);
161         cancel.setOnClickListener(new View.OnClickListener() {
162             public void onClick(View view) {
163                 dialog.cancel();
164             }
165         });
166 
167         return dialog;
168     }
169 
170     //Este método construye el diálogo emergente necesario para elegir el ejercicio del grupo en HosActivity
171     public Dialog createHostExerciseDialog(final HostActivity activity, final TrainApplication application) {
172         Log.i(TAG, "createHostExerciseDialog()");
173         final Dialog dialog = new Dialog(activity);
174         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
175         dialog.setContentView(R.layout.host_exs_dialog);
176         final ListView listAdapter = (ListView)  dialog.findViewById(R.id.exercisesList);
177 
178         //Hacemos uso de ExercisesListMaker al ser una lista personalizada
179         ExercisesListMaker adapter = new ExercisesListMaker(activity, listaEjercicios);
180         listAdapter.setAdapter(adapter);
181 
182         listAdapter.setOnItemClickListener(new ListView.OnItemClickListener() {
183             public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
184                 String ej = listAdapter.getItemAtPosition(position).toString();
185                 application.hostSetChannelExercise(ej);
186                 dialog.cancel();
187             }
188         });
189 
190 
191         return dialog;
192     }
193 
194     //Este método construye el diálogo emergente necesario para introducir la descripción del grupo en HosActivity
195     public Dialog createHostDescriptionDialog(HostActivity activity, final TrainApplication application) {
196         Log.i(TAG, "createHostNameDialog()");
197         final Dialog dialog = new Dialog(activity);
198         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
199         dialog.setContentView(R.layout.host_desc_dialog);
200 
201         final EditText description = (EditText)dialog.findViewById(R.id.hostDescChannel);
202         description.setOnEditorActionListener(new TextView.OnEditorActionListener() {
203             public boolean onEditorAction(TextView view, int actionId, KeyEvent event) {
204                 if (actionId == EditorInfo.IME_NULL && event.getAction() == KeyEvent.ACTION_UP) {
205                     String desc = view.getText().toString();
206                     application.hostSetChannelDescription(desc);
207                     dialog.cancel();
208                 }
209                 return true;
210             }
211         });
212 
213         Button okay = (Button)dialog.findViewById(R.id.hostDescOk);
214         okay.setOnClickListener(new View.OnClickListener() {
215             public void onClick(View view) {
216                 String desc = description.getText().toString();
217                 application.hostSetChannelDescription(desc);
218                 dialog.cancel();
219             }
220         });
221 
222         Button cancel = (Button)dialog.findViewById(R.id.hostDescCancel);
223         cancel.setOnClickListener(new View.OnClickListener() {
224             public void onClick(View view) {
225                 dialog.cancel();
226             }
227         });
228 
229         return dialog;
230     }
231 
232     //Este método construye el diálogo emergente necesario para iniciar el grupo en HosActivity
233     public Dialog createHostStartDialog(final HostActivity activity, final TrainApplication application) {
```

File - DialogBuilder.java

```
234         Log.i(TAG, "createHostStartDialog()");
235         final Dialog dialog = new Dialog(activity);
236         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
237         dialog.setContentView(R.layout.host_start_dialog);
238 
239         Button yes = (Button)dialog.findViewById(R.id.hostStartOk);
240         yes.setOnClickListener(new View.OnClickListener() {
241             public void onClick(View view) {
242                 application.hostStartChannel();
243 
244                 //Se establece una breve espera momentánea para dejar que el nombre del grupo se difunda
245                 SystemClock.sleep(500);
246 
247                 application.setMyGroup(true);
248                 if(activity.checkPrivate()){
249                     application.hostSetChannelPrivate(true);
250                 }
251                 application.chatSetChannelName(application.hostGetChannelName());
252                 application.chatSetChannelExercise(application.hostGetExercise());
253                 application.chatJoinChannel();
254                 dialog.cancel();
255             }
256         });
257 
258         Button no = (Button)dialog.findViewById(R.id.hostStartCancel);
259         no.setOnClickListener(new View.OnClickListener() {
260             public void onClick(View view) {
261                 dialog.cancel();
262             }
263         });
264 
265         return dialog;
266     }
267 
268     //Este método construye el diálogo emergente necesario para detener el grupo en HosActivity
269     public Dialog createHostStopDialog(HostActivity activity, final TrainApplication application) {
270         Log.i(TAG, "createHostStopDialog()");
271         final Dialog dialog = new Dialog(activity);
272         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
273         dialog.setContentView(R.layout.host_stop_dialog);
274 
275         Button yes = (Button)dialog.findViewById(R.id.hostStopOk);
276         yes.setOnClickListener(new View.OnClickListener() {
277             public void onClick(View view) {
278                 application.chatLeaveChannel();
279                 application.chatSetChannelName("");
280                 application.chatSetChannelExercise("");
281                 application.hostStopChannel();
282                 dialog.cancel();
283             }
284         });
285 
286         Button no = (Button)dialog.findViewById(R.id.hostStopCancel);
287         no.setOnClickListener(new View.OnClickListener() {
288             public void onClick(View view) {
289                 dialog.cancel();
290             }
291         });
292 
293         return dialog;
294     }
295 
296     //Este método construye el diálogo emergente al pulsar el botón Borrar historial de HistoricActivity
297     public Dialog createHistoricDeleteDialog(final HistoricActivity activity, final DBHandler db){
298         final Dialog dialog = new Dialog(activity);
299         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
300         dialog.setContentView(R.layout.historic_delete_dialog);
301 
302         Button yes = (Button)dialog.findViewById(R.id.historicDelOk);
303         yes.setOnClickListener(new View.OnClickListener() {
304             public void onClick(View view) {
305                 db.dropHistoric();
306                 activity.onResume();
307                 dialog.cancel();
308             }
309         });
310 
311         Button no = (Button)dialog.findViewById(R.id.historicDelCancel);
```

File - DialogBuilder.java

```
312         no.setOnClickListener(new View.OnClickListener() {
313             public void onClick(View view) {
314                 dialog.cancel();
315             }
316         });
317 
318         return dialog;
319     }
320 
321     //Este método construye el diálogo emergente en caso de producirse un error en la red P2P
322     public Dialog createAllJoynErrorDialog(Activity activity, final TrainApplication application) {
323         Log.i(TAG, "createAllJoynErrorDialog()");
324         final Dialog dialog = new Dialog(activity);
325         dialog.requestWindowFeature(dialog.getWindow().FEATURE_NO_TITLE);
326         dialog.setContentView(R.layout.alljoyn_error_dialog);
327 
328         TextView errorText = (TextView)dialog.findViewById(R.id.errorDescription);
329         errorText.setText(application.getErrorString());
330 
331         Button yes = (Button)dialog.findViewById(R.id.errorOk);
332         yes.setOnClickListener(new View.OnClickListener() {
333             public void onClick(View view) {
334                 dialog.cancel();
335             }
336         });
337 
338         return dialog;
339     }
340 }
341 
```

File - VideoFragment.java

```java
package org.proyecto.ricardo.p2ptrain;

import android.app.Fragment;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageButton;
import android.widget.VideoView;

//Clase encargada de inicializar el fragmento de video del ejercicio en la pantalla de chat
public class VideoFragment extends Fragment {
    private VideoView videoView;
    private ImageButton botonPlay;
    private ImageButton botonStop;
    public MediaPlayer mediaPlayer;

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState){
        View fragmentView = inflater.inflate(R.layout.video_layout, container, false);

        videoView = (VideoView) fragmentView.findViewById(R.id.vistaVideo);

        videoView.setOnPreparedListener(new MediaPlayer.OnPreparedListener() {
            @Override
            public void onPrepared(MediaPlayer mp) {
                mediaPlayer = mp;
            }
        });

        botonPlay = (ImageButton) fragmentView.findViewById(R.id.botonPlay);
        botonPlay.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v)
            {
                if (mediaPlayer != null){
                    mediaPlayer.start();
                    ((ChatActivity) getActivity()).msjPlay();
                }
            }
        });

        botonStop = (ImageButton) fragmentView.findViewById(R.id.botonStop);
        botonStop.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v)
            {
                if (mediaPlayer != null){
                    mediaPlayer.pause();
                    ((ChatActivity) getActivity()).msjPause();
                }
            }
        });
        return fragmentView;
    }
}
```